

PhoenixSim 1.0 User Manual

Photonic and Electronic Network Integration and eXecution Simulator

Gilbert Hendry and Johnnie Chan

{gilbert, johnnie} @ee.columbia.edu

Lightwave Research Laboratory

Columbia University

New York, NY

Contents

1	Overview	5
1.1	Background	5
1.2	Photonics in Networks-on-Chip	5
1.3	What is PhoenixSim?	6
1.4	OMNeT++	6
1.5	How to Get PhoenixSim	6
1.6	The Rest of This Document	8
1.6.1	Conventions	8
1.7	Required Reading	8
2	Simulator Structure	9
2.1	Folder Hierarchy	9
2.2	Big Picture	9
2.3	Processing Plane	10
2.3.1	Network Interfaces (NIFs)	11
2.3.2	Applications	12
2.4	Network Plane	16
2.4.1	Electronic Router	16
2.4.2	Photonic Devices	20
2.4.3	Hybrid Router	22
2.5	IO Plane	22

2.5.1	DRAMsim	22
2.5.2	DRAM-LRL	24
2.5.3	Core to Memory Mapping	25
2.6	Addressing	26
2.6.1	Defining Network Addresses	29
2.6.2	Address Translation	29
2.7	Statistics and Results	29
2.7.1	Result file naming	31
2.7.2	Registering statistics	31
3	Communication Protocols	32
3.0.3	Communication Stack	32
3.1	Processing Plane	33
3.1.1	Application event timing	33
3.1.2	NIF-Processor Communication Protocol	34
3.1.3	NIF-Network Communication Protocol	34
3.2	Network Plane	35
3.2.1	Electronic Router Interactions	35
3.2.2	Photonic Device Interactions	36
3.3	IO Plane	37
3.3.1	DRAMsim	37
3.3.2	DRAM-LRL	37
4	Network Design	40
4.1	Electronic Networks	40
4.1.1	Electronic Router	40
4.1.2	Electronic Channels	41
4.1.3	Top-Level Parameters	41

4.2	Circuit-switched Photonic Networks	42
4.2.1	Insertion Loss - Bandwidth tradeoff	43
4.2.2	4×4 Switch Designs	45
5	Current Networks	46
5.1	Electronic Networks	46
5.1.1	Electronic Mesh	46
5.1.2	Electronic Mesh - Circuit Switched	46
5.1.3	Electronic Torus	47
5.1.4	CMesh	47
5.1.5	Flattened Butterfly	47
5.2	Hybrid Circuit-Switched Photonic Networks	48
5.2.1	Photonic Torus	48
5.2.2	Nonblocking Torus	48
5.2.3	Photonic Mesh	48
5.2.4	TorusNX	49
5.2.5	Square Root	50
6	How To ...	51
6.1	Create my own photonic device	51
6.2	Create my own Network Interface (NIF)	52
6.3	Create my own electronic router	52
6.4	Create my own hybrid router	52
6.5	Create my own network/topology	52
6.6	Run my own application/application model	53
6.7	Measure something with custom statistics	53
7	Appendix	54
7.1	List of all OMNeT Parameters	54

7.2	ORION Parameters	57
-----	----------------------------	----

Chapter 1

Overview

This document is used to completely describe PhoenixSim to a person wishing to use or modify it.

1.1 Background

The concept of a NoC has brought about a change in the way microprocessors are being designed today. Exploration of the architecture, implementation, and impact of NoCs in future generations of processors is paramount to being able to scale performance while maintaining (or achieving) power efficiency. However, actual implementations have generally lagged behind the multitude of proposed designs. Leading instantiations of today's NoCs include Tiler's TILE and TILE-Gx chips [21] (a 10×10 mesh) and the Cell Processor's Element Interconnect Bus [10] (a centrally-arbitrated circuit-switched 12-port redundant ring). These would be considered "baseline" implementations in today's NoC research community.

There are many reasons why chips coming off today's assembly lines don't have the complicated NoC architectures that have been the subject of many-a-research. The heart of the matter lies in the fact that most microprocessors, general purpose or otherwise, don't have *or need* very many cores. Programming many-core architectures is a serious problem that has not been sufficiently addressed as of yet, and therefore software generally can't make use of too many cores. In addition, memory bandwidth is generally the performance bottleneck anyway.

But fear not, researcher of the network-on-chip: there is hope. The problems just mentioned will be solved one day, and the fruit of our labor will be required.

1.2 Photonics in Networks-on-Chip

Using light as the medium for transmitting data has been around for some time now, gaining widespread acceptance in the long-distance telecommunications industry for a few important reasons:

- Light travels a long way (kilometers) without having to be regenerated, whereas electronics must drive a capacitive wire every inch of the way. Think of it like this: optics is like shooting a gun, electronics is like plowing snow. This can be a huge advantage in terms of energy consumption.
- Light travels at the speed of, well, light. Actually about $(2/3)c$ in optical single-mode fiber [6], but that's still fast.
- Light signals of different wavelengths can all cram into one waveguide or fiber. We call this Wavelength Division Multiplexing (WDM), and results in much higher bandwidth density (read: smaller cable bundles).

Here at the Lightwave Research Lab, we believe optics is a good candidate for on-chip communication for basically the same reasons as it was adopted into telecomm. However, to make optics viable for integration into microprocessors, the devices must be CMOS compatible. Silicon nanophotonics is a field that has emerged researching optical devices manufactured in silicon towards this end.

1.3 What is PhoenixSim?

PhoenixSim (Photonic and Electronic Network Integration and eXecution Simulator) was originally designed to allow us to investigate silicon nanophotonic NoCs taking into account component's physical layer characteristics. Because photonics often requires electronic components around it for control and processing, we also incorporated models of some typical electronic network components. We ended up with a simulation environment that is suited to investigate both electronic and photonic NoCs.

1.4 OMNeT++

PhoenixSim is built on OMNeT++ [22], an environment for creating any event-driven simulator. Making PhoenixSim event driven means it's relatively efficient for what it does. However, many of the events that occur in the simulator are on a clock-cycle granularity, which means it does not sacrifice too much accuracy. OMNeT supplies functional libraries, mechanisms for instantiating components, managing parameters, and executing batches of simulations. With the release of OMNeT 4.0, we also now have an Eclipse-based IDE.

When programming in OMNeT, code is written in C++ that defines how components behave. Files are all automatically compiled into a single executable, which can then be run as a simulation, which is supported both through a GUI and on command-line (for batch runs).

OMNeT also defines its own NED language for defining the interface and structure of components. Familiarity with this language is necessary for PhoenixSim users who wish to add components or networks, though using the examples we currently have, it should not be a problem. We are also working on a photonic device layout tool which will decouple the user from having to write NED files.

Since OMNeT 4.0 is now Eclipse-based, PhoenixSim is compatible in both Windows (with MinGW) and Linux. We prefer Windows, as a not-entirely-correctly configured Linux platform may cause frustrating unexpected OMNeT shutdowns, which may or may not have been fixed at the time of this writing.

1.5 How to Get PhoenixSim

The following are step-by-step instructions for setting up PhoenixSim on your local machine to run simulations, or modify the code.

Setting up OMNeT++:

1. Download OMNeT++. It can be found at <http://www.omnetpp.org>.
2. Install OMNeT++. Extract it somewhere on your computer. For Windows, you'll have to open the MinGW environment (`mingwenv.cmd`), and compile in that. In the extracted folder, run `./configure` and `make` to compile OMNeT. The `./configure` may end with some warnings. Ignore the ones that could not detect any of the following: BLT, MPI Akaroa (all optional).

There may be some instructions for including paths, etc. Go ahead and do that (bash profile for Linux, Environment Variables in Windows).

3. Once OMNeT is compiled, open it up (omnetpp on the command line), and go to your workspace.

Getting PhoenixSim:

1. Download the PhoenixSim zip file from <http://lightwave.ee.columbia.edu/phoenixsim>, and *don't* extract it.
2. First, in the OMNeT environment, right click on the project explorer area. Select New → OMNeT++ project...
3. Name the project PhoenixSim, and use whatever location you want. Click Finish.
4. You now should have a basic project. You will be doing a project Import in the Eclipse OMNeT environment, if you are familiar with how that works in Eclipse. Right click on the top-level project name, and select Import...
5. The Import wizard should come up. Select General → Archive. Click Next.
6. For the From Archive field, browse to wherever you saved the zip file.
7. Make sure this folder appears in the left-hand pane, and that the check-box is checked. Click Finish.
8. You should now have all the code in the project. Select the top-level project again, and hit Cntrl-B to build.
9. You should not have any errors (red x's in the project explorer), and a PhoenixSim.exe executable should now be there.

Running a simulation:

1. Open the Parameters folder.
2. Click on omnetpp_sample.ini.
3. Press Cntrl-F11, which is used to either run a simulation with a particular .ini file, or run the last one used.
4. The OMNeT GUI should come up, and prompt you to select a Configuration. Do so, and Click ok.
5. In the main simulation window, click the Express run button (or hit F7).
6. Close the simulation, and view the results in the results folder. You successfully just tested the connectivity of the network you chose by sending a small message from every core to every other core.

1.6 The Rest of This Document

Chapter 2 is an introduction to how the simulator works, and some modeling details. Chapter 3 some of the interactions between different components in the simulator. Chapter 5 describes the networks that are currently modeled and come with PhoenixSim as examples. Chapter ?? describe things that researches might typically want to use PhoenixSim for, such as measuring the power and performance of a network design. Chapter 6 describes typical ways in which the simulator is extended to support additional components and functionality to suit the needs of a particular simulation.

1.6.1 Conventions

We will try to use the following visual conventions when describing different parts of the simulator to make it easier for the reader

- PLANE
- *Class*
- OMNeT Compund Module
- *OMNeT Simple Module* (they are also classes)
- **OMNeT Message**
- Parameter

1.7 Required Reading

You may want to become familiar with some things before reading this document. First of all, become at least familiar with OMNeT and the NED language (www.omnetpp.org). The easiest way to do this may be to look at one of our network examples.

For information on circuit-switched photonic networks, you can read [9, 19]. Other work describing some network-level physical considerations can be found in [2, 3]. Information about some of the photonic devices we model can be found in [4, 12, 14].

Chapter 2

Simulator Structure

This chapter explains the overall structure of the simulator, and some details on how components are modeled.

2.1 Folder Hierarchy

The components you will find in the PhoenixSim code are organized into various folders depending on their function. The following enumerates and describes the contents of these folders:

- chipComponents - contains definitions of some useful components used in NoCs
- electronicComponents - contains the components used in electronic routers, including the ORION power model [25]. Also contains all *Arbiters*, which are used to define new routing functions.
- ioComponents - contains components used in the IO PLANE, which is currently limited to DRAM. Contains DRAMsim [24] for modeling contemporary DRAM subsystems. Also contains dramNET, our DRAM models based on the simple timing requirements of DRAM devices.
- parameters - contains default parameter values for all components
- photonic - contains all the models of the photonic devices including with PhoenixSim
- processingPlane - contains components necessary for modeling traffic generation
- simCore - contains core functions, such as message definitions and statistics
- topologies - contains the network topologies that come with PhoenixSim

2.2 Big Picture

Most of the computing systems that can be modeled in PhoenixSim look like Figure 2.1. The PROCESSING PLANE consists of many cores executing applications, which produce communication events to the NETWORK PLANE. The IO PLANE can consist of anything off-chip, including DRAM modules or inter-chip communication.

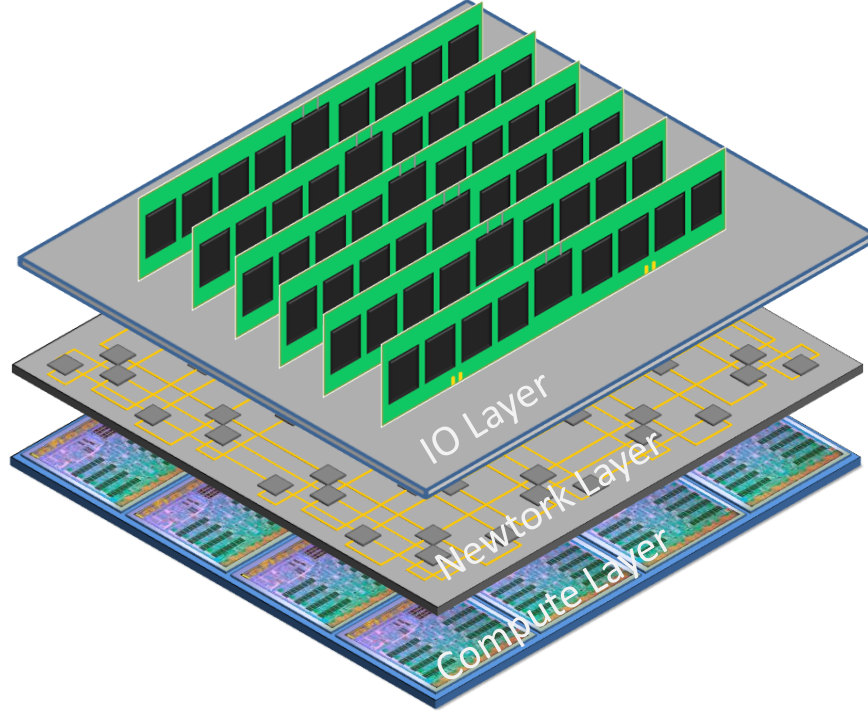


Figure 2.1: Simulation consisting of 3 planes: Processing, Network, IO

2.3 Processing Plane

The PROCESSING PLANE consists of multiple *Processors*, which represent cores or other processing elements, each having an instance of an *Application* class which dictates how communication events are generated. The structure of the PROCESSING PLANE varies slightly depending on if and how *concentration* is implemented, but generally looks like Figure 2.2. An instance of a *Processor* exists to model each independent processing core in the CMP. A Network InterFace (NIF) translates a *Processor*'s communication request event into the network-specific protocol needed to complete it.

The idea of multiple cores sharing a single logical network node has been established as probably a good idea for CMPs with a large number of cores to reduce network resources [11]. This can actually be accomplished in different ways, shown in Figure 2.3.

The first is network-side concentration. This involves modifying the network switches and routers to accept multiple injection/ejection points, such as increasing the number of ports (radix) in an electronic router, as was done in [11]. Network-side concentration requires no changes in the PROCESSING PLANE, because each *Processor* still has one *NIF*, which it uses to interface to the increased-radix routers.

The second way to accomplish core concentration is using a concentrated gateway. This method leaves the network routers in tact, while using a switch to connect local *Processors* with a single shared *NIF*. This second method is preferred for photonic networks because it saves on the number of modulators and detectors needed, and thus power and area.

The basic tradeoffs between the two methods are in the number of *NIFs*, having a *Processor*

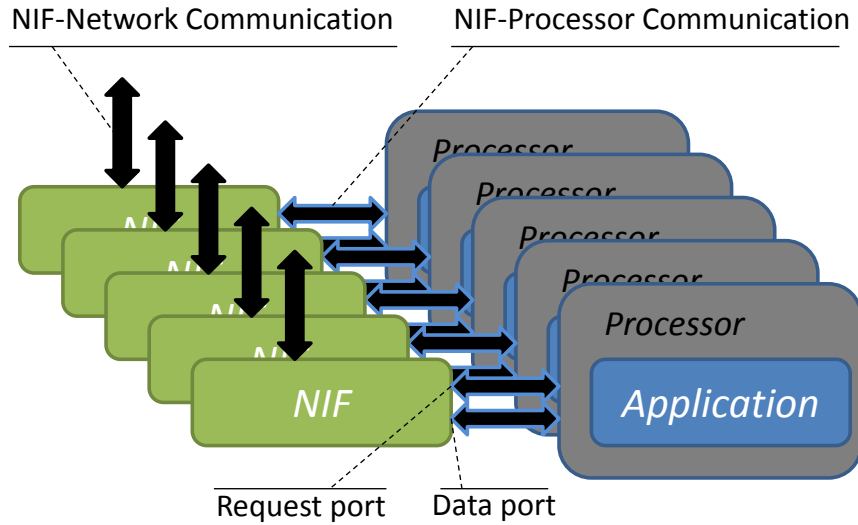


Figure 2.2: Structure of the PROCESSING PLANE

Router (bufferless), and radix of the network routers. Either way, which method you choose effects how you make the connections from the PROCESSING PLANE to the NETWORK PLANE, so be aware of the differences.

Both versions of the processing plane are in `processingPlane/processingplane.ned`. Network-side concentration is the regular `ProcessingPlane` module. The concentrated gateway version is the `ProcessingPlane_ProcRoute` module, which uses the `ProcessorRouter` to implement the electronic crossbar that feeds into the shared NIF.

2.3.1 Network Interfaces (NIFs)

The *NIFs* in PhoenixSim provide a way for us to decouple the design of the network with subsystems connected to the network, such as the PROCESSING PLANE. A *NIF* initiates new communication on the network by sequentially calling 4 functions which are pure virtual in the *NIF* class, and therefore required by all subclasses:

- `request()` - called when a communication request arrives from the non-network side of the *NIF*. Returns true if communication should continue.
- `prepare()` - called after `request()` returns true. Usually creates a new message to be sent on the network. Returns true if communication should continue, which is almost always the case.
- `send()` - called after `prepare()` returns true. Responsible for sending the prepared message. Again, usually always returns true.
- `complete()` - called after `send()`. Performs anything that is required after the transmission has completed. Returns the amount of time that the *NIF* should wait before attempting to start a new transmission.

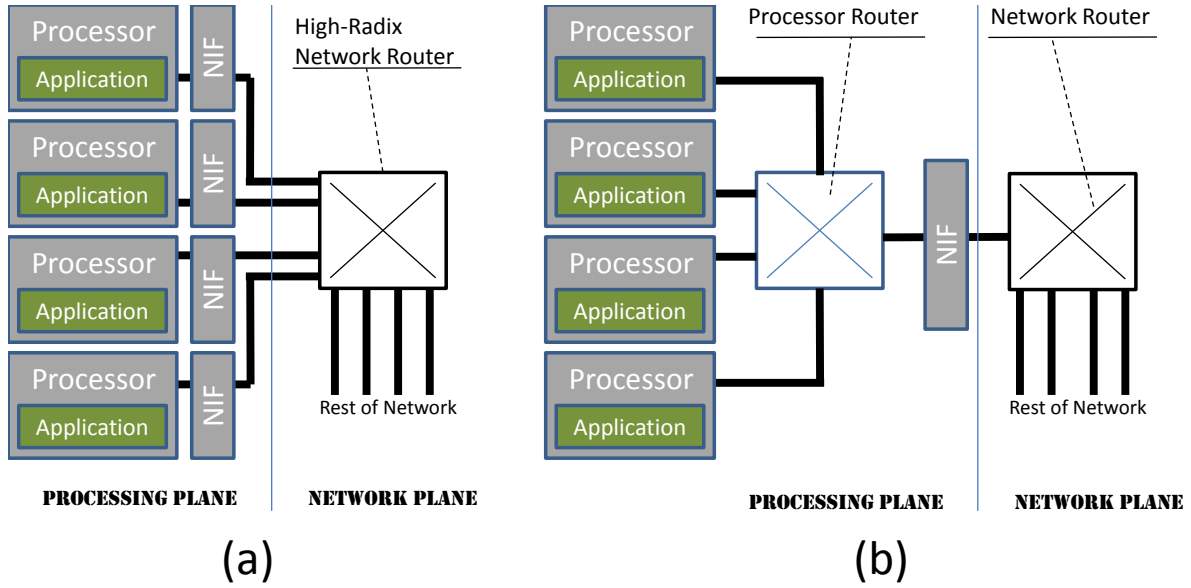


Figure 2.3: Core concentration (a) Network-side (b) Concentrated gateway

Figure 2.4 shows the hierarchy of existing *NIF*s. Currently, we have two other abstract classes that are available to inherit from: *NIF_Packet_Credit* and *NIF_Circuit*. The *NIF_Packet_Credit* class contains some useful functions for interacting with credit-based packet-switched networks, such as standard electronic ones. The *NIF_Circuit* class assumes a packet-switched network which controls a circuit-switched data plane, and therefore further builds on *NIF_Packet_Credit*.

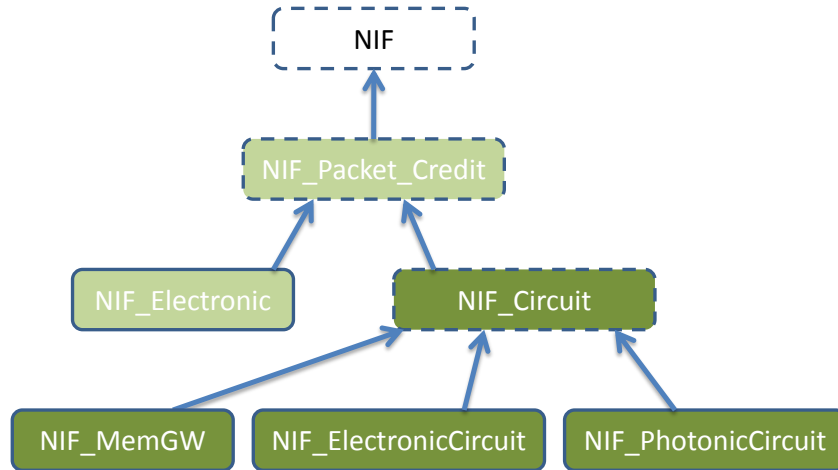


Figure 2.4: Hierarchy of current NIFs. Abstract classes shown outlined with dashes.

2.3.2 Applications

The application model in PhoenixSim works by informing the *Processor* that owns it what the next communication event will be, and how long the *Processor* should wait before sending it

(i.e. how long it takes to process the code that generates the communication event). As discussed later in Section ??, an *Application* can respond to any of 4 events that the *Processor* informs it of: first message, data arrived, sending data, and message sent. A separate instance of an application runs on each *Processor*, which enforces any communication to take place on the network. This is a good way to make sure that an application model does not have any information that would not actually be available to it on a real system.

Parameters to an *Application* are passed through general OMNeT parameters called appParam1, appParam2, appParam3, and appParam4, which are all of type *double*. There is also an appParam5 of type *string* which may be used. These parameters are made available to the *Application* superclass, and therefore to all its subclasses. It is up to the specific instance of the *Application* how to use these parameters.

Now, we will describe all the applications that come with PhoenixSim, found in processing-Plane/apps/.

All2All

In *All2All*, every core sends one or more messages (depending on appParam2) to every other core, as fast as possible. This is usually used as a basic functionality test, to see if the network routes packets correctly to every destination. This application can also be used to find the worst case insertion loss in a photonic network, as all possibilities will be tested. If DRAM is connected to the chip, All2All will also test all *Processor* to DRAM module combinations.

Table 2.1: All2All Application Parameters

Parameter	Purpose
appParam1	Packet size, in bits.
appParam2	Number of times to repeat the whole process.
appParam3	Not used.
appParam4	Not used.

Random

The *Random* application specifies the usual random traffic. Each core randomly selects a destination (uniformly), and messages arrive as a Poisson process. The next message for a core is scheduled when the previous one has been sent to the *NIF*, which means that under heavy network load, many messages could potentially become queued up at the *NIF*. The Random application model does not backpressure because of this, but keeps generating messages.

Table 2.2: Random Application Parameters

Parameter	Purpose
appParam1	Mean inter-arrival time, in seconds.
appParam2	Size of each packet, in bits.
appParam3	Number of messages each core should send before stopping.
appParam4	Not used.

DRAM test apps

There are 3 applications to test *Processor* to DRAM communication. They are One2One, One2All, and All2One, and do exactly what their names imply. One2One can be used to test the zero-load latency of a memory access. One2All specifies that one *Processor* communicates with every DRAM module, and tests if all memory modules are accessible. All2One specifies that all *Processors* communicate with a single memory module, testing the contention mechanism. All three applications use basically the same parameters.

Table 2.3: DRAM Test Apps Parameters

Parameter	Purpose
appParam1	Number of messages to send (All2One, One2One only).
appParam2	Read (1) or Write (0).
appParam3	Size of the packet, specified as $2^{\text{appParam3}}$ bits.
appParam4	Which core is sending (One2One only).

DRAM Random

This application model is basically the same as the Random application, except that no *Processor* - *Processor* takes place, only *Processor* - DRAM. This is useful for testing a sort of shared-memory like programming model.

Table 2.4: DRAM Random Application Parameters

Parameter	Purpose
appParam1	Mean arrival times, in seconds.
appParam2	Size of each packet, specified as $2^{\text{appParam3}}$ bits.
appParam3	Number of messages to send before stopping.
appParam4	Boolean, (0): use statically-mapped memory module, (1): use random memory module.

DataFlow

The DataFlow application model is made to approximate a graph/dataflow/streaming/pipelined execution model, where each *Processor* computes some part of the total computation on a "piece" of data, and passes it on to his neighbor, and repeats the same computation for the next "piece" of data that arrives. So a large dataset is broken up into small data pieces and pipeline through the network. Figure 2.5 shows the communication patterns. Basically data flows from a memory module across each row and column, to each memory module on the opposite sides, in both directions. This assumes a network which has memory access points on the ends of each row and column.

FFT

Out FFT model is based on the Cooley-Tukey algorithm [5], where the FFT is computed in stages, each core sending their piece of computed data to someone else. An 8-core version of this algorithm is shown in Figure 2.6. Its fairly easy to see how it would look with many cores.

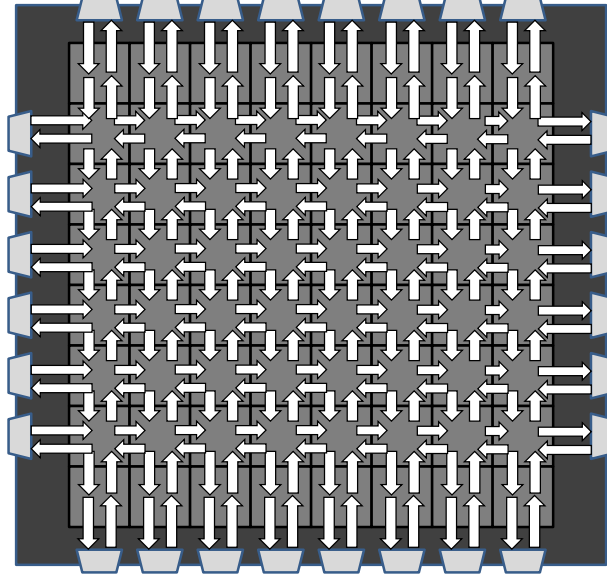


Figure 2.5: Dataflow application communication patterns.

Table 2.5: DataFlow Application Parameters

Parameter	Purpose
appParam1	Time for processing a piece of data, in seconds.
appParam2	Size of each piece of data, specified as $2^{\text{appParam2}}$ bits.
appParam3	Number of pieces of data to stream.
appParam4	Not used.

The first stage is the main computation stage, and takes the longest. All the following stages consist of each core combining another core's results with its own, and take about an order of magnitude less time than the computation stage. The amount of time required for the main computation stage and combination stages must be supplied as parameters.

You can use any method you want to determine what these numbers should be, but we refer to Frigo and Johnson's work in characterizing FFT computations [7]. We use these numbers and information about the cores they used to get our computation times.

For example, they report that a Pentium M core takes 39.32ms to compute the FFT on 256k samples. We also know that the Pentium M core was 84mm^2 in 130nm technology. Using classical scaling, the core would be 5.25mm^2 in 32nm technology (which is what we usually simulate at, though it's possible to do others. See Section 7.2 for details on using different technologies). This puts our total die size at 336mm^2 for 64 cores, which is reasonable.

For the combination stages, we must extrapolate from the computation stage, given that we know the complexity of the computation stage is $5k\log(k)$, and the complexity of the combination stage is simply $5k$. We can then solve a simple proportion to infer that the combination stage would take 2.18ms on the Pentium M.

Finally, we plug our numbers into the correct parameters and let her fly.

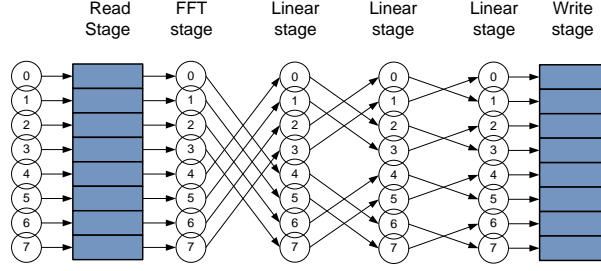


Figure 2.6: FFT computation per the Cooley-Tukey algorithm.

Table 2.6: FFT Application Parameters

Parameter	Purpose
appParam1	Each core's data size, specified as $2^{appParam1}$ bits.
appParam2	Time for main computation stage, in seconds.
appParam3	Time for combination stages, in seconds.
appParam4	Bitmap indicating which stages to perform. 0 bit - read, 1 bit - fft, 2 bit - write.

FFT_stream

The FFT_stream application executes the same communication patterns as the FFT. However, instead of transferring large messages all at once, it breaks them up into smaller messages, in a data-flow / streaming paradigm. Take a look at the code, you'll get the idea.

2.4 Network Plane

The main purpose of PhoenixSim is in the NETWORK PLANE, which consists of routers and switches to transfer data from one point to another. First, let's go over the devices and components that we use to build routers, both electronic and photonic. Let's first look at electronic components.

2.4.1 Electronic Router

Our electronic router model is your basic store-and-forward packet-switched router. A high level diagram can be seen in Figure 2.7. Functionally, we model it as having a 3-stage pipeline: message arrival and request, arbitration, and switch traversal. Currently, we use credit-based Bubble flow control [17], though we will be looking to implement more sophisticated mechanisms in the future.

For power modeling, ORION [25] is used for nearly every electronic component in PhoenixSim. In most cases, we use ORION's record(...) and report(...) functions to record the energy for each individual event as they happen, as opposed to their stat.energy(...) functions, which try to calculate energy dissipation based on an activity factor.

We will now go into detail on how each part is modeled.

RouterInport

The microarchitecture for the *RouterInport* can be seen in Figure 2.8. A single inport has a separate SRAM array for each virtual channel. Minor control logic interacts with the RouterArbiter, handling requests and grants. The *ORIONArray* class is used to model the power for each virtual

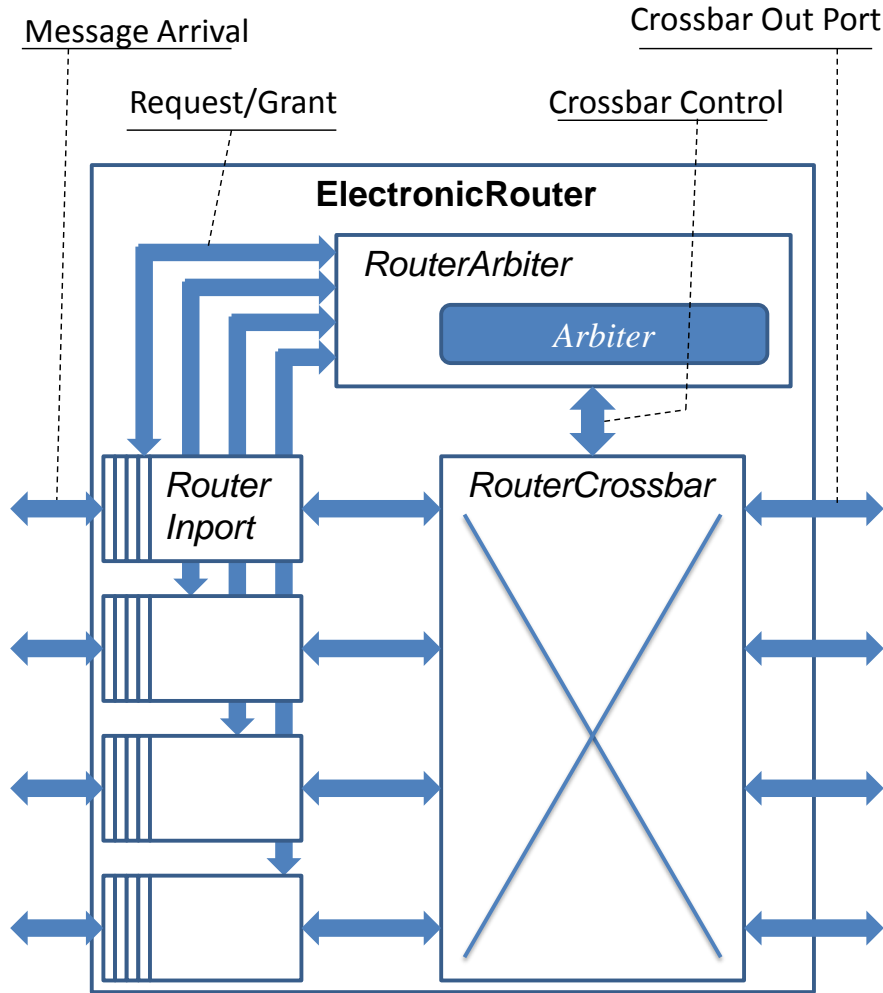


Figure 2.7: Structure of Electronic Router.

channel buffer.

RouterCrossbar

The *RouterCrossbar* is modeled as a dumb crossbar: it accepts commands from the *RouterArbiter* to set up input-output connections, and accepts messages arriving at an input, forwarding them to the correct output. The *ORION_Crossbar* is used to model the power when messages traverse it, modeling the actual switching of input-output connections and 50% probability of bit-changes on each line.

RouterArbiter

The *RouterArbiter* implements three main functions: routing, contention, and device setup. Because these things can be different depending on the network, we allow the user to specify his own subclass of *Arbiter*. The *Arbiter* class contains the basic algorithm for ensuring that a message is correctly routed.

Figure 2.9 shows the class hierarchy for arbiters. The following are the main abstract arbiter

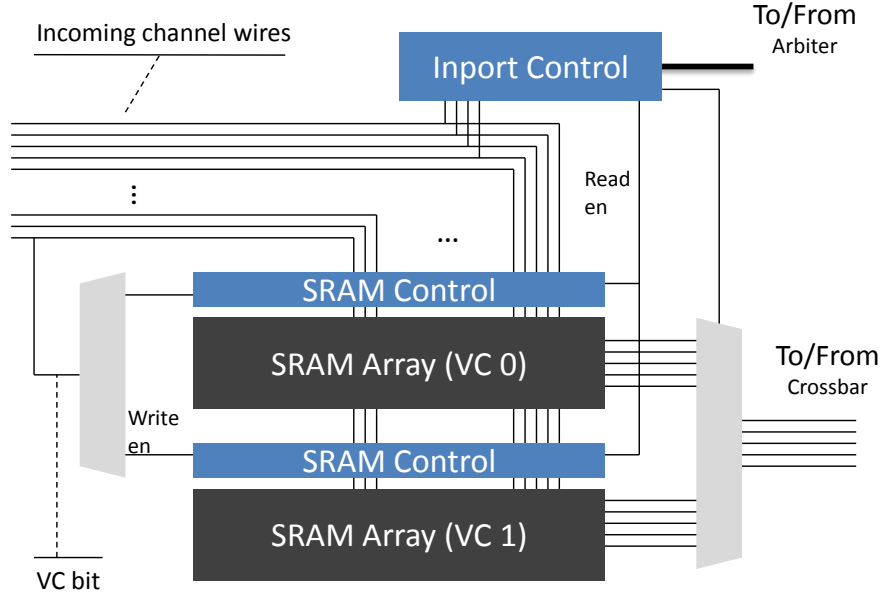


Figure 2.8: **RouterInport** microarchitecture.

classes which capture different functionality:

- *ArbiterCanRoute* - parses and compares the *NetworkAddress* of the router and destination of the message. See Section 2.6 for more details on addressing in PhoenixSim.
- *Arbiter* - main superclass, contains main round-robin loop for examining waiting messages.
- *ElectronicArbiter* - implements Bubble credit-based flow control
- *PhotonicArbiter* - implements logic for handling various path-setup messages. See Figure 3.3 for more details on path setup.
- *PhotonicNoUturnArbiter* - checks for U-turns, which is not allowed in some photonic switch designs.
- *Photonic4x4Arbiter* - implements device control logic for the different designs of 4x4 photonic switch. See Section 4.2.2 for more details.

For now, all of the arbiters for the current networks we have implemented inherit from one of these abstract classes (See Chapter 5 for the networks that come with PhoenixSim). See Section 6.3 for details on how to create a new arbiter for your network.

ElectronicChannel

The *ElectronicChannel* module models optimally-repeated intermediate/global electronic wires in parallel connecting routers together. The channel uses a basic formula for computing the time it takes to transmit data:

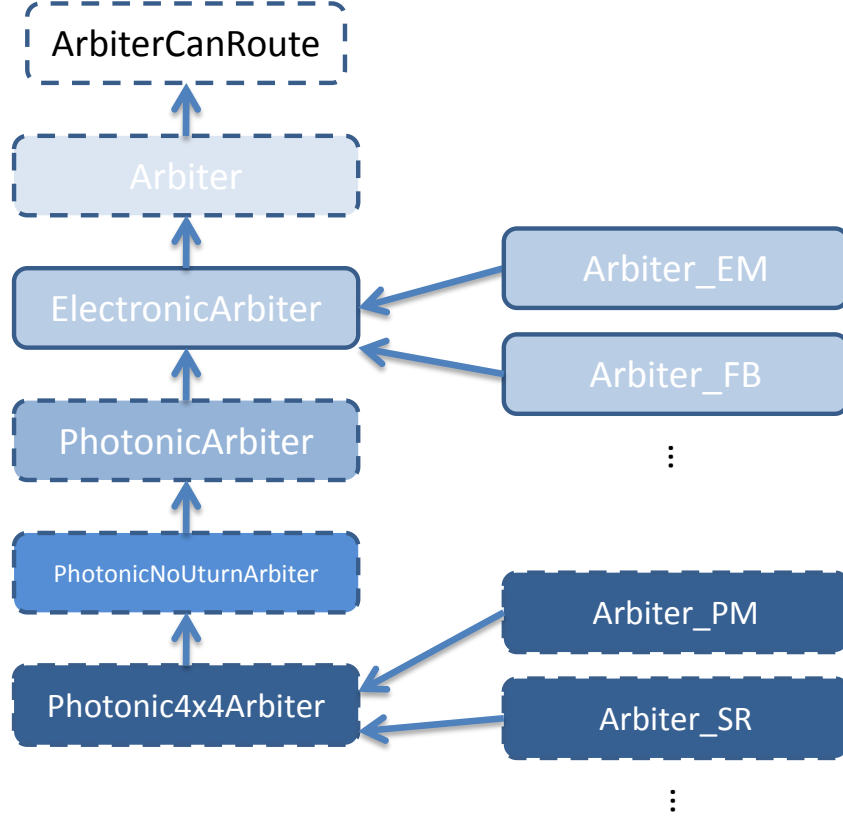


Figure 2.9: Arbiter class hierarchy.

$$t_{transmission} = clockPeriod * (numRepeaters + messageSize / (channelWidth)) \quad (2.1)$$

ElectronicChannel uses *ORION_Link* to model its power, and obtain the optimal number of repeaters based on the length. The length of the channel is specified as number of inter-router spaces and number of router widths. Router width is calculated in the *RouterStat* module (see below), and inter-router spaces are calculated as the width of a core minus the router width, assuming one router per core.

RouterStat

The *RouterStat* module is included in the *ElectronicRouter* to calculate both the router area and the router's clock power, both as estimated by ORION. The inports, arbiter, and crossbar all send an OMNeT message to the *RouterStat* on simulation startup, indicating their parameters which *RouterStat* uses to calculate area and clock power. ORION provides an estimate of total area, and *RouterStat* assumes that the router is square.

Virtual Channels

We've mentioned our support for virtual channels. Currently, they are implemented using separate physical buffers. To avoid packets from a large application-level messages from arriving at a destination out of order, we do *not* allow a packet to change virtual channels in a network. Once a virtual channel id is assigned to a packet, it stays that way until it gets to where it's going. The one exception is when virtual channels are used as priority channels in circuit-switching control (not going to get into that). An electronic network expert might cite this as a severe implementation flaw, and he might be right, as electronic network performance could be enhanced using fancy flow control techniques, etc. We'll be adding support for that kind of thing in the future.

2.4.2 Photonic Devices

Our library of photonic devices comprises of all photonic technologies required to generate, control, and receive an optical signal. This library of devices can be used to create any number of switch fabrics and network topologies. Our efforts in building a framework for describing these devices has required us to strike a balance between a physical accuracy and system level performance simulation. On one hand, we want to be able to model the devices in a physically accurate way without requiring a full FDTD simulation, on the other hand, we also want to be able to simulate the devices operating in a network environment to produce meaningful system-level performance results. This has resulted in the development of a Basic Element abstraction for describing all photonic devices.

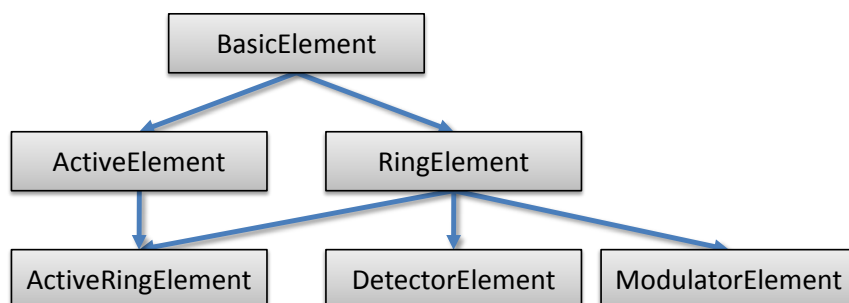


Figure 2.10: Hierarchy of the photonic modeling classes.

The current class hierarchy is shown in 2.10. The BasicElement base class abstracts all the photonic characteristics common to all photonic devices. Practically, this describes characteristics of insertion loss and latency. Beyond the BasicElement class, we provide subclasses which are used to describe more specific types of devices. Currently this is used to describe passive devices such as straight waveguides, bending waveguides, waveguide crossings, and couplers.

The RingElement class inherits from the BasicElement class, and provides a means to describe the resonant behavior of rings (and in fact any resonator devices, such as Mach-Zehnders). In essence, RingElements are just BasicElement devices that exhibit wavelength dependency. This means that an optical signal coming in at a certain wavelength may behave differently from a second signal that has a different wavelength. RingElement is currently used as the inherited class for ring-based filters.

The ActiveElement class also inherits from the BasicElement class, but additionally provides mechanisms for describing devices that can exhibit multiple states. This class should not be con-

fused with active physical devices, which would generally describe a component that requires input power to function. ActiveElements describe devices that have multiple logic states, such as a switch that can be 'on' or 'off'. For example, the 'on' state might indicate that a switch is in a cross state, while the 'off' state will indicate that the switch is in a bar state.

While our currently library does not contain any strictly ActiveElement devices, we have a number of devices that belong in the ActiveRingElement group, which inherits from both the RingElement and ActiveElement class. This includes the 1×2 and 2×2 photonic switching elements (PSEs).

Lastly, we have also included the DetectorElement and ModulatorElement classes which are special classes for injecting and detecting optical messages. Specifically, our library currently only contains detector and modulator devices that are based on ring resonators, therefore the classes can in actuality be more accurately described as RingDetectorElement and RingModulatorElement.

Next we describe some of the devices that have been modeled and are currently included in our library. This described set of components describes a complete set of devices that can be used to form a complete optical network.

Waveguides

Waveguides are the optical wires that connects all photonic devices together. Optical signals experience attenuation (*i.e.* insertion loss) as they propagate through a waveguide which is due to fabrication imperfections such as side-wall roughness.

Waveguide Bends

Bending is a necessary requirement for properly routing optical paths and creating topologies. Additional sidewall roughness due to the bend in the waveguide as well as the optical mode leaking out of the waveguide cross section results in additional losses. This is largely dependant on the the size and radius of the bend.

Waveguide Crossings

Due to the planar nature of on-chip photonics, waveguide crossings are an inevitable consequence of non-trivial networks. Because the number of crossings in a network can be quite significant, a designer should pay careful consideration to the number of crossings found in the network and the limits that can be created depending on the assumed losses due to the crossings.

Couplers

Couplers provide the functionality necessary in simulating the interface between two disjoint waveguides. An example of such an interface is an optical signal that must travel between the an on-chip silicon waveguide and an external optical fiber.

Ring Filters

Ring filters are devices based on ring resonators that selectively switch an optical signal, dependent on its wavelength. Because rings exhibit multiple resonant peaks in its spectral profile, a ring filter can be designed to tuned to multiple optical wavelengths, providing a mechanism to passively switch multiple signals through wavelength-selective switching. [15]

Ring-Based Broadband Switches

Ring devices can also be designed to include thermal or electro-optic control mechanisms for active control of the device. This can be referred to as a ring switch, since it is not necessary to use wavelength selectivity to control a signal's path, but instead can be directed by signaling the switch itself. Additionally, since wavelength selectivity is no longer used to control the path, the entire spectrum can be used for the data signal, allowing the use of high bandwidth wavelength division multiplexed (WDM) signals. [23] [13]

Ring Modulators

Ring modulators, in principle, operate in a similar way to electro-optic ring switches. In the 'off' state, an optical signal will pass a ring unobstructed, whereas in the 'on' state, an optical signal will be coupled out of the waveguide, thereby removing the light. In this way, a series of bits can be encoded onto an optical stream for transmission. [26]

Photo-Detectors

Our currently photo-detector model uses a Germanium detector, which has been experimentally fabricated and is CMOS compatible. Strictly, the photo detector itself does not require a ring, however since the detector is a relatively broadband device, a filter is required before a signal can be properly received. Therefore an easy way to select the proper wavelength for detection is to include a ring filter just before the detector device. [?]

2.4.3 Hybrid Router

Our HybridRouter module consists of an ElectronicRouter which controls a switch, seen in Figure 2.11. Using multiple hybrid routers, if we connect the electronic routers together and the switches together, we effectively form a circuit-switched network where the electronic routers are the control plane, and the switches are the data plane.

The HybridRouter module has two important parameters: `elRouter` and `optSwitch`, which specify the arbiter type for the electronic router, and the name of the switch which implements the module interface *PhotonicSwitch*, respectively. These two parameters allow you to use the HybridRouter module in different circuit-switched networks by simply writing a new *Arbiter* and implementing a new *PhotonicSwitch*.

2.5 IO Plane

The structure of the IO PLANE very much depends on the simulation being run. Currently, we put our DRAM models into this separate plane, but you could put anything there that represents off-chip components (other chips, outside networks, etc). We will briefly describe our DRAM device and control models, and how they interact with the network.

2.5.1 DRAMsim

There are two models for the DRAM subsystem included with PhoenixSim. The first is DRAMsim [24], a well-known DRAM simulator from the University of Maryland. This model is usually used to simulate contemporary DRAM subsystems. It contains the usual DRAM devices and control you would expect to see in today's computers (well, maybe yesterday's). We're not going to describe the whole thing, so you can check out the website for more details <http://www.ece.umd.edu/dramsim/>. Typically, we use DRAMsim when simulating packet-switched electronic NoC's. Because packet-

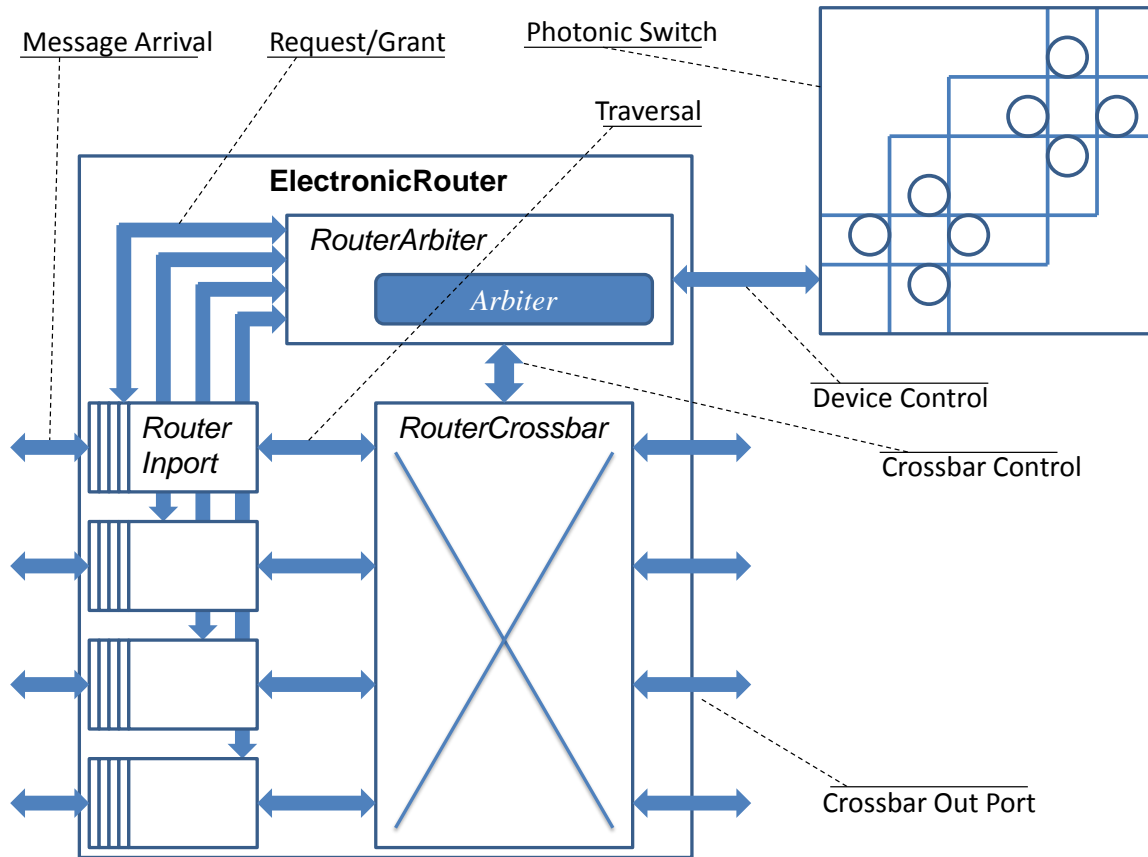


Figure 2.11: Hybrid router consisting of a photonic switch, controlled by an electronic packet-switched router.

switched networks use relatively small packets that must fit into network buffers, they are closely aligned with today's memory access mechanisms (*i.e.* accessing single cache lines at a time).

DRAMsim contains all the models necessary for a complete DRAM subsystem, including the memory controller, transaction queue, bank, chip, rank, address translation, and many different configurations for control policies, as well as performance and energy consumption.

We've slightly modified DRAMsim to fit into PhoenixSim. First, we refactored it into C++. Why it was originally written in C is a mystery to us, as it lacks serious organization and characteristics of good software. We've also wrapped it into an OMNeT module, called *cDramModule*. Figure 2.12 shows the structure of the modules we typically instantiate when using DRAMsim.

While the *ElectronicWire* model is used for on-chip wires, *ElectronicOpad* is used for off-chip ones. The *cDramModule* wrapper module does a couple of things:

- Interact with the *NIF* attached to it. In this way, the *cDramModule* implements the same interface that the *Processor* uses. See, we told you the *NIF-Processor* protocol would come in handy.
- Queue up transactions. DRAMsim comes with a transaction queue model which it handles.

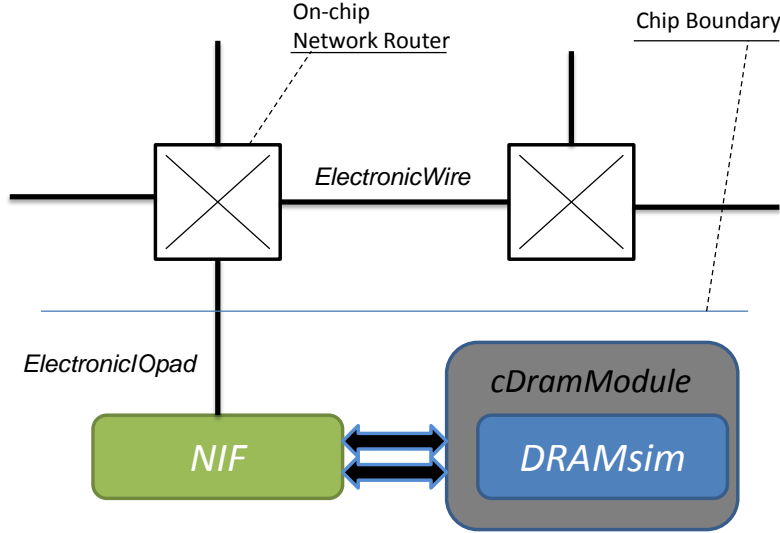


Figure 2.12: **DRAMsim** used in a packet-switched network.

The *cDramModule* wrapper attempts to insert incoming transactions into DRAMsim. If the transaction queue is full, the wrapper is responsible for providing back-pressure to the *NIF*.

- Managing large messages. In PhoenixSim, we allow large messages to be broken up into small packets so they can fit into the router buffers. However, if a read transaction arrives at the *cDramModule* which requests a large amount of data, the *cDramModule* actually issues many small read transactions to DRAMsim, because that's what DRAMsim expects: cache-line size accesses.

Again, we use DRAMsim for modeling the memory subsystem for electronic networks, because DRAMsim was made to model today's shared-memory small accesses coming from a small number of shared caches, which can approximate how a packet-switched NoC might behave.

2.5.2 DRAM-LRL

The second model is one that we have developed at LIGHTWAVE RESEARCH LABORATORY, which we call DRAM-LRL, which is used to model a hypothetical memory subsystem design in the context of circuit-switched networks. Full details can be found in [8]. DRAM-LRL was made to fit into PhoenixSim more efficiently by staying event-driven, as opposed to DRAMsim which models every cycle. DRAM-LRL is further simplified by the fact that it is made to be used in circuit-switched networks. Figure 2.13 shows the components we model in DRAM-LRL. It shows a Circuit-Accessed Memory Module (CAMP), which has a central *OCM_Transceiver* which performs O-E-O conversion, and controls the address and data local bus usage according to the stage of the transaction. Only one transaction need be sustained at a time, due to circuit-paths having exclusive access to the module.

Figures 2.14 and 2.15 show the chip-side memory controller, which mitigates memory bank contention and converts transactions into DRAM device commands, as well as the data plane switch that enables end-to-end communication between cores and DRAM modules. Figure 2.16 shows the flowchart for how the memory controller works.

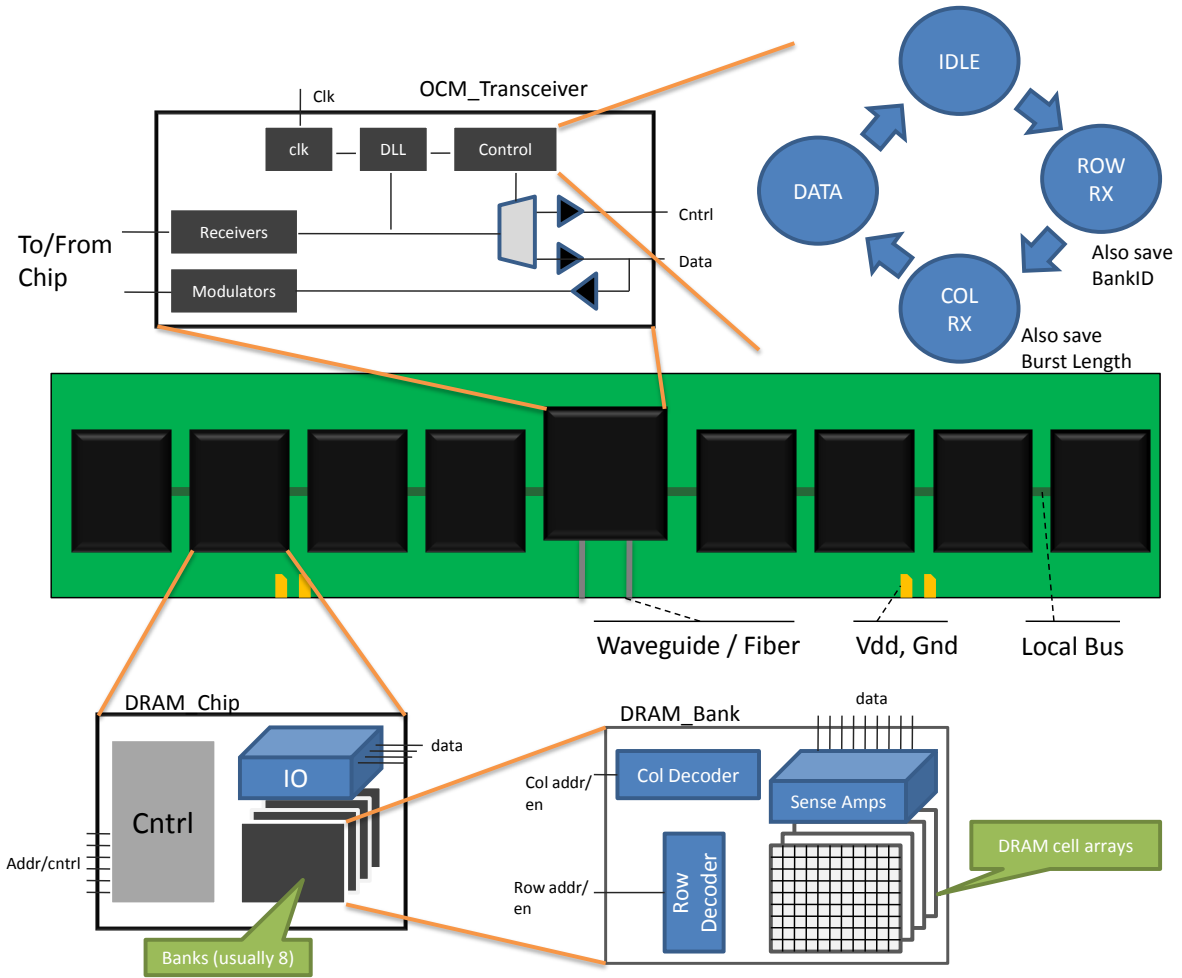


Figure 2.13: CAMM structure in DRAM-LRL.

2.5.3 Core to Memory Mapping

Each *Processor* in PhoenixSim can be viewed as having its own local memory space. We use a class called *DRAM.Cfg* to dictate how a *Processor* locates its memory space. An *Application* should therefore refer to the *DRAM.Cfg* instance in the *Application* superclass to find out where it should go for memory accesses, if the programming model has such a thing as "local" memory spaces. *DRAM.Cfg* has two main functions:

- `getAccessId(int coreId)` - returns the id of the network node where this *Processor*'s memory space is attached to.
- `getAccessCore(int dramId)` - returns the id of the network node that the `dramId` memory bank is attached to.

Using these functions, a *Processor* can ask the *DRAM.Cfg* where its local memory address space is, and how to get there from here. Figure 2.17 shows how cores are usually mapped to memory gateways, assuming one memory gateway per peripheral network node.

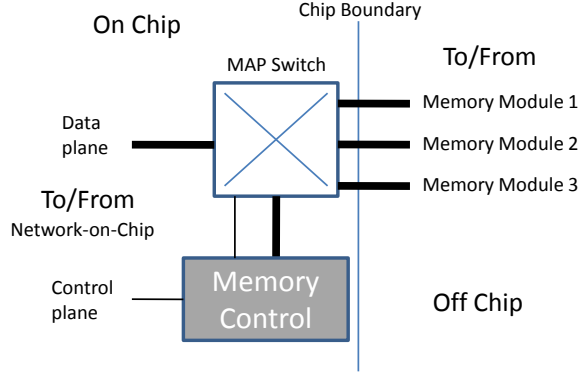


Figure 2.14: Memory Access Point.

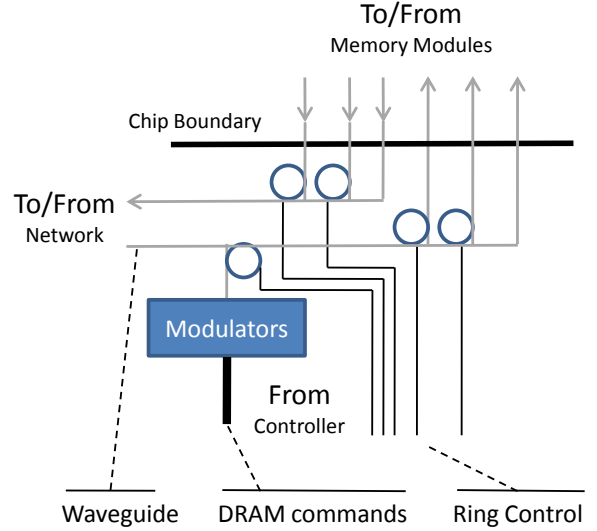


Figure 2.15: MAP Switch.

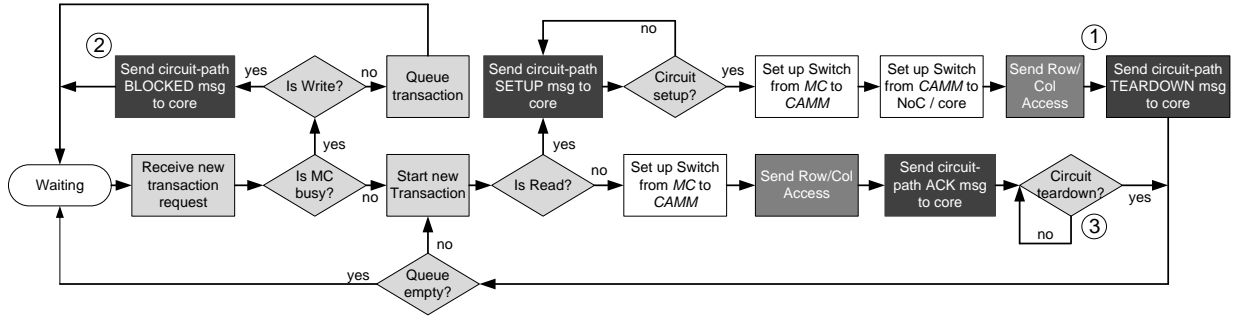


Figure 2.16: Flowchart of DRAM-LRL memory controller.

For some networks, it may be useful to define other mappings. In that case, just override *DRAM_Cfg* and instantiate it in *Processor*, so that the *Application* can use it.

2.6 Addressing

So, how are cores in the network addressed? This was kind of a problem if we wanted to support many different kinds of networks. Sure, you could give them just some random unique integer, but it makes it easier on the routing logic if the id's make sense.

So we came up with a hierarchical addressing scheme. Furthermore, we've made it so you can define the structure of your addresses however you want. We use the concept of address *domains*, much like an IP address. In PhoenixSim, the left-most address is considered the "top", and the right-most the "bottom". Lets do a simple example.

Consider the 2×2 network in Figure 2.18, where each router is concentrated with 4 cores attached to it. The format of our addresses is:

$$NET.MEM.PROC \quad (2.2)$$

where NET is the network domain, or the routers. MEM is the memory gateway domain.

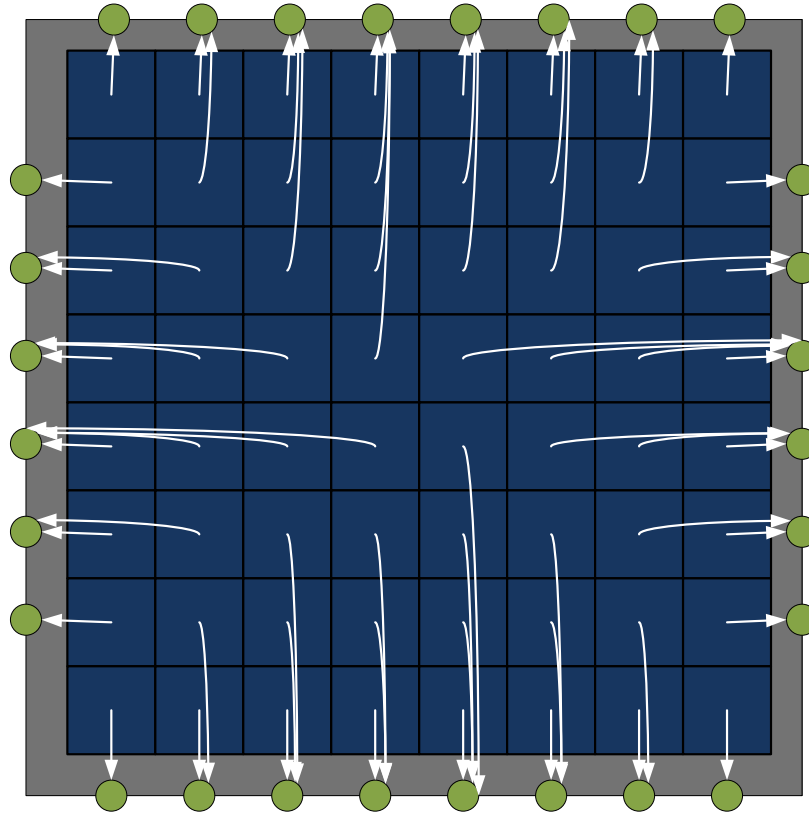


Figure 2.17: Default core-memory map.

PROC is the processor domain. Note how the addresses are assigned in Figure 2.18. In addition, each *Arbiter* has a *level*, which specifies which domain it is in. Since the routers are the "top" level, they only have the NET domain defined.

The process for routing up and down the address domains is defined in *ArbiterCanRoute*. Recall from Section 2.4.1 that every *Arbiter* must inherit from *ArbiterCanRoute*, because that class contains the mechanism for parsing the addresses. When defining an *Arbiter*, you can implement three functions:

- `route(...)` - decides where to forward the message when in the same address domain. This is required.
- `getUpPort(...)` - decides which port to go to when we need to go up a domain. Defaults to `route(...)`.
- `getUpDown(...)` - decides which port to go to when we need to go down a domain. Defaults to `route(...)`.

When a message reaches an arbiter, it looks at the destination address, starting with the top level. Depending on which address does not match the arbiter's address, and which level the arbiter is will dictate which function gets called.

Lets look at the communication labeled Example 1 in Figure 2.18. Core 0.0.1 wants to send to Core 1.0.3. Here's how it works:

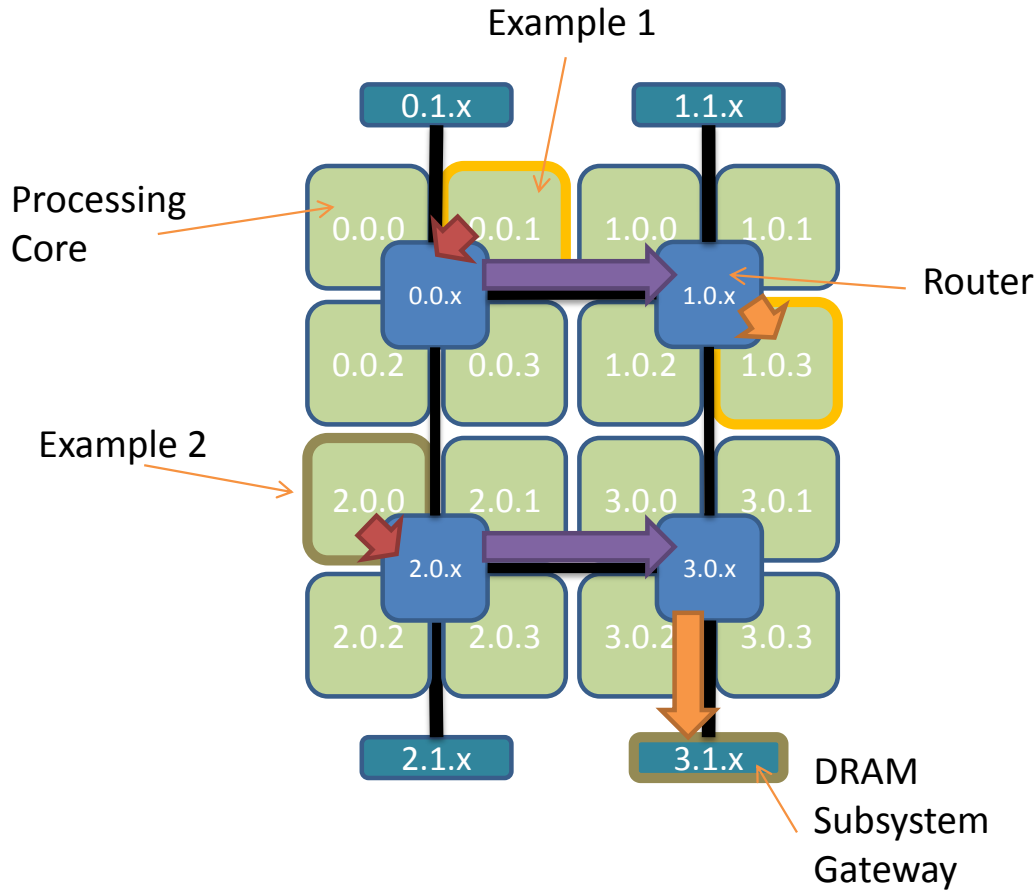


Figure 2.18: Example of how addressing and routing works.

1. Core 0.0.1 must decide where to forward his message. It starts with the top level, NET. The destination is 1.0.3. The NET domain is 1 for the destination, and 0 for the sender. They are different. Also, the level of the Core is PROC. Since the first difference was encountered at a level higher than the Core's level, it calls `getUpPort(...)`. This is indicated by the red arrow in Figure 2.18.
2. Now Router 0.0.x must decide where to forward the message. It compares its address (0.0.x) to the destination (1.0.3). Again, the difference occurs in the NET domain. Since the Router is in the NET domain, it calls `route(...)`, which sends it East. This is indicated by the purple arrow.
3. Router 1.0.x must now decide where to forward the message. It compares its address (1.0.x) to the destination (1.0.3). The first difference occurs in the PROC domain, which is lower than the router's level (NET). Therefore, the `getDownPort(...)` function is called. This is indicated by the orange arrow.

Example 1 was for Core-Core communication, basically bypassing the MEM domain. Example 2 illustrates how this domain can be used. Core 2.0.0 wants to send a message to DRAM gateway

3.1.x. Note that the gateway itself does not have an address. Core 2.0.0 sends a message there by indicating the NET-domain address of the router connected to it. In this case, Router 3.0.x. It then specifies a 1 for the MEM domain, indicating that when the message gets to Router 3.0.x, it should call `getDownPort(...)` in response to the difference between it's MEM address (0) and the destination's MEM address (1). The arbiter must check for this case in the `getDownPort(...)` function.

2.6.1 Defining Network Addresses

In PhoenixSim, you are free to define your own address format. The above examples used the NET.MEM.PROC format, which is common for a flat network structure. Indeed, any network could use this format. But you can make your life easier when writing your arbiters by using different formats.

By default, when you instantiate a `processingPlane` module, it tacks on the MEM.PROC domains. Typically, you define the network domains using the `networkProfile` parameter. This is done using a string which looks like

$$** .networkProfile = "name1.name2.;[N1].[N2]."$$
 (2.3)

where `name1` and `name2` are domain names, and `[N1]` and `[N2]` are the number of nodes contained within these domains. The number of domains you can have is arbitrary. For instance, the format used above in our examples would look something like this:

$$** .netorkProfile = "NET.;" + string(X * Y) + "."$$
 (2.4)

where you've already defined `X` and `Y` as the number of network nodes in the `x` and `y` coordinates.

2.6.2 Address Translation

There's only one more detail you need to know. The *Application* classes use integers to denote other processing cores. This is done on purpose, because an application model should not have to worry about the network it's running on. The distribution of an application should be a logical construct, not a physical one to decouple how we write and distribute an application from the hardware it's running on.

So, we need a mechanism to translate from logical unique integers to addresses in the network. This is done using the *AddressTranslator* class, found in `processingPlane/addressTranslation/`. If you define your own addressing format, you need to write your own *AddressTranslator*. Don't worry, it's easy. All you have to do is convert an incoming **ApplicationData** to a `NetworkAddress`.

The *AddressTranslator_Standard* class is used for the regular NET.MEM.PROC format. Look at that to see an example.

2.7 Statistics and Results

The point of the simulator is, of course, to measure statistics to tell us what's going on, from energy to performance. We've set up a few mechanisms to make things easier. Every network should contain one and only one *Statistics* simple module. This module creates the results files, and is where all statistics are registered. This code is found in `statistics/statistics.cc`, and is where any result file changes must be made (other than just adding new statistics to the current single

result file).

Figure 2.19 shows the logical organization of statistics in PhoenixSim.

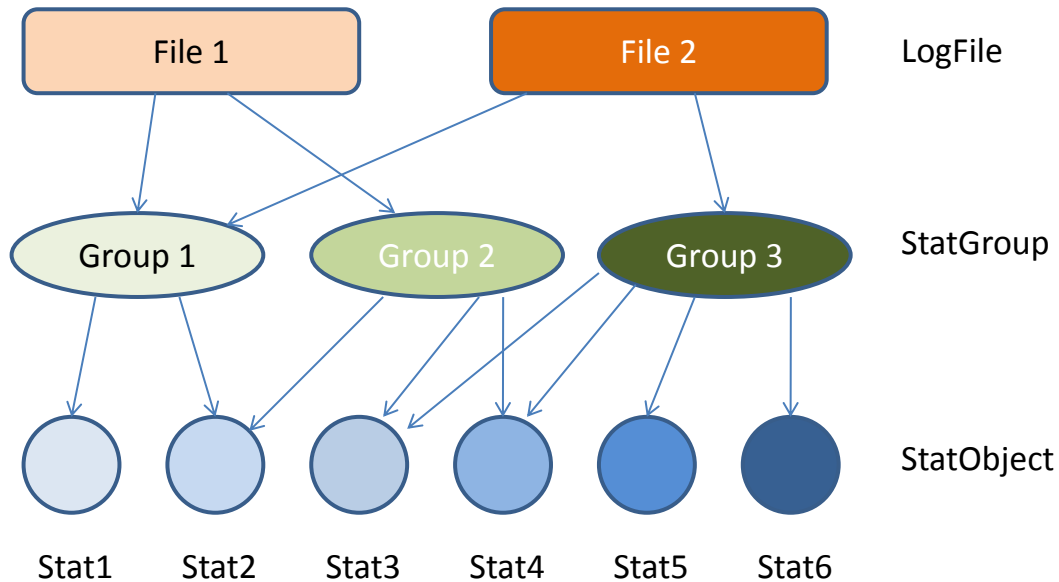


Figure 2.19: How statistics work.

There are three classes you could be aware of:

- *LogFile* - this describes a results file, which is a collection of StatGroups.
- *StatGroup* - this defines how to organize different StatObjects. For instance, whether just to list them all or add them all up and report the sum.
- *StatObject* - this defines how something is actually measured. For instance, taking the time average or just total of all measurements.

Basically, a LogFile can contain any number of StatGroups, which define what goes in the file. A StatGroup is a collection of different statistics. They basically end up getting their own section in the result file. When a StatObject is created in the simulation, it must first register itself with the central Statistics module. When that happens, any StatGroup that exists can pick it up. A StatGroup must know a few things (passed to the constructor) to gather only the right StatObject:

- Name - the section header to write in the file
- Range - the range of StatObject types that should be included in this group. For instance, ENERGY_STATIC - ENERGY_LAST specifies all energy-related statistics. See StatObject.h for types.
- Filter - a string as an additional filter. Only statistics coming in specifying they are part of this group will be added.

Take a look at Statistics::Statistics() to see the StatGroups that are instantiated. Finally, when you want to measure something, you create a StatObject, and call its track(...) function to record the value.

2.7.1 Result file naming

One thing we've done is provide automated file naming. Currently, there is just one file that is produced when a simulation is run. There are three parameters that are of interest here:

- logDirectory - the place to put the results file(s). This is relative to *where you are running the simulation from*. This is important, depending on whether you are using the OMNeT GUI or command-line tools.
- networkName - give the network a name.
- customInfo - anything else you want in the file name.

The results file that results will be [logDirectory]simStats_[networkName]_[customInfo].csv.

2.7.2 Registering statistics

When you want to measure something, first you need the right type of StatObject. The following are the ones currently defined:

- Count - counts the number of times this object is created. This is useful for counting the number of a particular type of components get created in your network.
- MMA - stands for Min Max Average. Calculates these stats for all values that are tracked.
- TimeAvg - reports the average over time for all values tracked.
- Total - sums up all the tracked values.
- EnergyEvent - sums up all discrete events that consume energy.
- EnergyOn - records the energy a component uses when it's considered "on".
- EnergyStatic - records the static power of the component at the beginning of the simulation, and reports it as total static energy at the end.

To enable a StatObject to be collected, ask the Statistics class, for instance:

```
StatObject* P_static = Statistics::registerStat("myLeakagePower",  
    StatObject::ENERGY_STATIC, "electronic");
```

returns a StatObject that can be used to record the electronic leakage power, which you would do by calling

```
P_static->track(0.014);
```

which would specify 14mW of leakage power.

Chapter 3

Communication Protocols

This chapter explains the interactions between various components.

3.0.3 Communication Stack

The groups of components in PhoenixSim use well-defined sets of messages to communicate to each other, much in the same way that a traditional communication stack works. Figure 3.1 shows how messages are encapsulated up the stack.

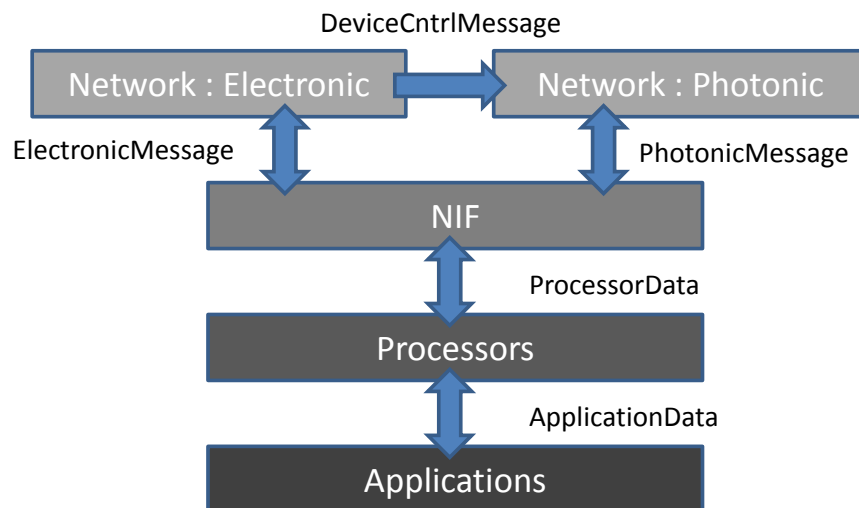


Figure 3.1: Message types in the communication stack

A *Processor* generates **ApplicationData** messages dictated by an instance of the superclass *Application* which it contains. It encapsulates these messages into **ProcessorData** messages and sends them to the *NIF* it is connected to. The *NIF* then handles these requests accordingly, depending on the network. Typically, **ElectronicMessage** messages are sent through electronic routers modeling data traffic as well as buffer acknowledgements. **PhotonicMessage**'s are sent through the photonic components of the network, and contain fields that are necessary for modeling physical-layer characteristics.

3.1 Processing Plane

Figure 3.2 shows typical communication in the PROCESSING PLANE between the *NIF*, *Processor*, and *Application*. In this example, the *Application* generated a first message, then waited until it received a message from another core.

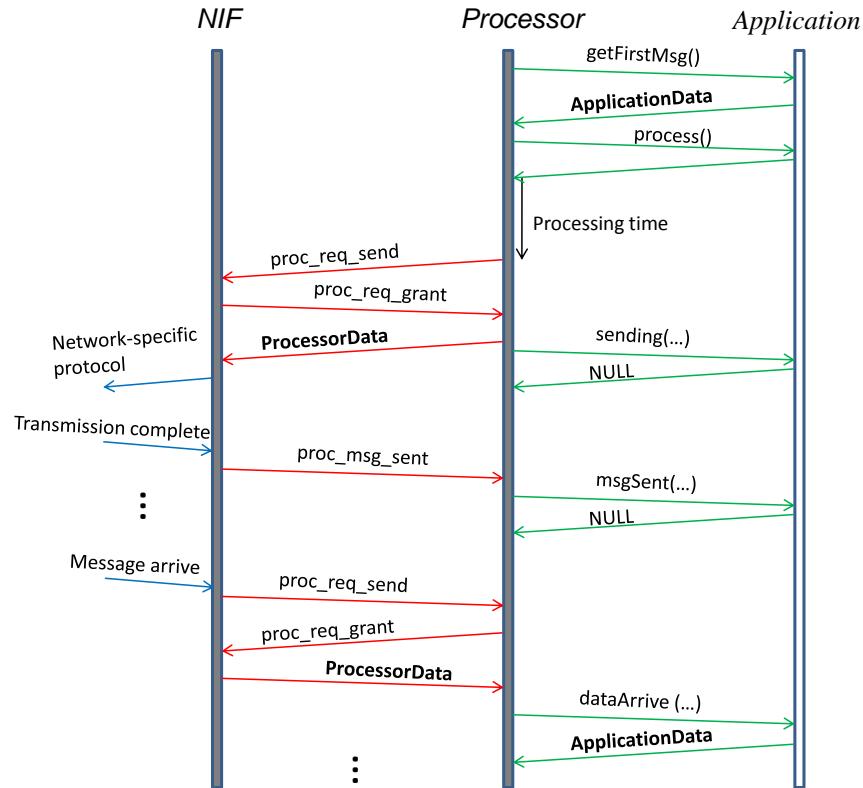


Figure 3.2: Communication between *Application*, *Processor*, and *NIF*

The communication events shown in Figure 3.2 are color coded by the interacting pairs. The following subsections describe the 3 main protocols in the PROCESSING PLANE: application event timing (green), *NIF-Processor* communication (red), and *NIF-Network* communication (blue).

3.1.1 Application event timing

Because an *Application* is a class instantiated by a *Processor*, the two communicate through the *Processor* invoking its *Application*'s functions, seen as the green arrows in Figure 3.2. An *Application* can respond to any of 4 events that the *Processor* informs it of:

- First Message - the first message of the application
- Data Arrive - any communication event that should occur after data arrives to this *Processor*
- Sending - another communication event that should occur when a message has been sent to the *NIF* to be sent

- **Message Sent** - a communication event that should occur after a message has completed sending to another *Processor*.

In each case, the application can return an **ApplicationData** that represents the next communication event. New applications can be written for PhoenixSim by defining the 4 functions listed above. See Section 2.3.2 for more details on the applications that come with PhoenixSim. See Section 6.6 for more details on creating new applications.

3.1.2 NIF-Processor Communication Protocol

The handshaking protocol for communicating between *NIF* and *Processor* is set up to allow different configurations of the PROCESSING PLANE. Any entity connected to a *Processor* or the processor side of a *NIF* must be able to accept and correctly handle **RouterCntrlMsg** messages with the following msgType that are incoming on the request port:

- **proc_req_send** - request to send **ProcessorData** on the data port
- **proc_req_grant** - it is ok to send the data

This protocol allows us to model when communication between a *Processor* and *NIF* may not be allowed, such as if a *NIF* transmit buffer is full because of network congestion. It also allows us model concentrated cores which share a single *NIF*. See Section ?? for more details on concentrated gateways.

Note that for a Concentrated Gateway, the handshaking between the *Processor* and *NIF* is intercepted by a **Processor Router**, a bufferless switch controlled by an arbiter. The **Processor Router** enables local *Processor-Processor* communication, and accessing the outside network. The *NIF-Processor* handshaking protocol comes in handy here because the **Processor Router**'s arbiter must handle contention between requests from the different *Processors* and the *NIF*. For circuit-switched networks, the **Processor Router** maintains circuits between a *Processor* and *NIF* until transmission across the network is complete.

3.1.3 NIF-Network Communication Protocol

Communication between a *NIF* and the network it is connected to depends on the kind of network. Generally, a new *NIF* must be written when implementing a different kind of network (See Section 6.2 for more details). Note that different "kinds" of networks does not mean topologies. Kinds of networks here may mean ones which use credit-based electronic routers, or hybrid circuit-switched photonic networks, or photonic wavelength-arbitrated networks. At the end of the day, a *NIF*'s job is to translate incoming messages to **ProcessorData** messages, which are then forwarded to a *Processor* via the mechanism discussed above in Section 3.1.2. The following paragraphs are some examples of *NIF*-network communication.

Electronic Networks. We assume some credit-based flow control mechanism for using fixed-size buffers in the electronic routers. This means that the *NIF* must be aware of the available buffer space in the router it is connected to. The *NIF* can send **ElectronicMessages** as long as it doesn't overrun the buffer. It can expect to receive **ElectronicMessages** back with the **ElectronicMessage::msgType** field set to **ElectronicMessageTypes::router_bufferAck**, to indicate that some of the buffer has been vacated. And that's it for packet-switched electronic networks.

Circuit-Switched Networks. Circuit-switched networks require a control mechanism to set up end-to-end circuit paths. This mechanism could lie in the same physical routers as the data plane (the circuit-paths), or separate (as in the case for photonics). Figure 3.3 shows an example of a path setup. In this example, the resources required at Router₂ were in conflict, and a *path_blocked* message is returned to the *NIF* that requested the path setup. Currently, a linear backoff period is implemented for congestion control. At the receiving end, when the *NIF* receives a *path_setup*, it returns a *path_ACK*, where the sending *NIF* can transmit the data on the data plane, and finally release the reserved network resources with a *path_tearardown* message.

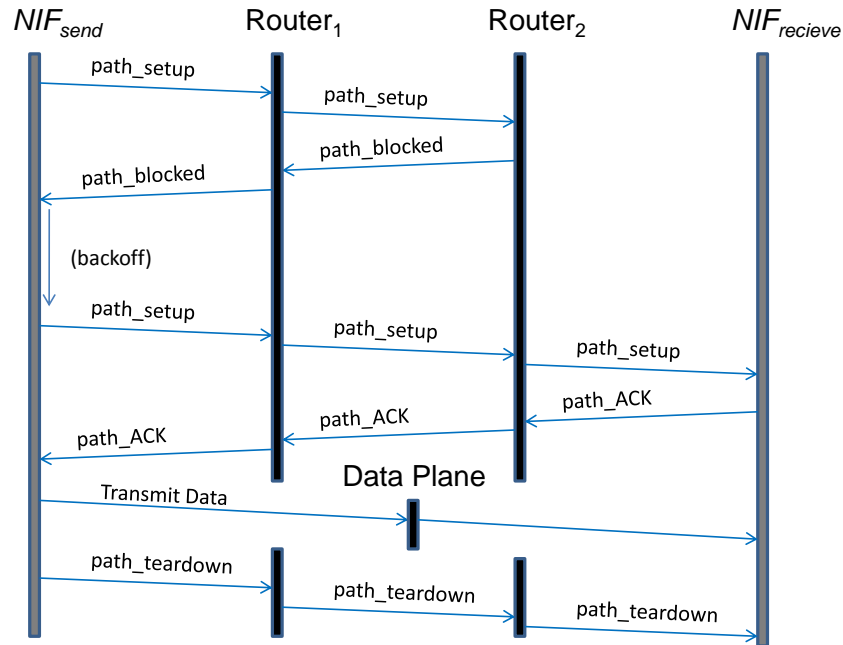


Figure 3.3: Path setup protocol.

3.2 Network Plane

This section briefly describes the interactions between the components that currently exist.

3.2.1 Electronic Router Interactions

Figure 2.11 shows a typical hybrid router in a photonic circuit-switched network, focusing mainly on the interactions between the components of the *ElectronicRouter*. These interactions are described as follows.

Message Arrival. An *ElectronicMessage* should arrive to a *RouterInport*, where its size is accounted for in the buffer. This triggers a *RouterCntlMsg* sent to the *RouterArbiter* to request its next hop.

Request/Grant. Interactions with the *RouterArbiter* come in the form of *RouterCntlMsg* messages. Route logic, photonic device control, and resource allocation are all in an instance of an *Arbiter* class, which is usually written when a new network requires different routing. See

Section 6.3 for details on how to write a subclass of *Arbiter*. The *RouterArbiter* calls its *Arbiter*'s functions to determine which inport receives a grant, and how to set up the crossbar for traversal.

Crossbar Setup. The *RouterCrossbar* is controlled from the *RouterArbiter* using **XbarCntrlMsg** messages, which simply specify an inport (connected to a *RouterInport*) and an output (connected to the outside).

Crossbar Traversal. When a *RouterInport* receives a **RouterCntrlMsg** that specifies it has been granted to send the message at the front of its queue, it forwards the **ElectronicMessage** to its output, which is connected to one of the *RouterCrossbar*'s inports. The inport also sends another **ElectronicMessage** set as a credit-acknowledgement, specifying that the buffer has emptied a little. The inport also examines the new front of its queue, and sends a new **RouterCntrlMsg** to the *RouterArbiter* if necessary.

Photonic Device Control. If the instance of *Arbiter* is one that needs to control photonic devices, it returns a **DeviceCntrlMsg** for each photonic device to its *RouterArbiter*, who forwards them to the photonic devices themselves, which are responsible for interpreting them.

3.2.2 Photonic Device Interactions

A signal traveling through photonic devices is modeled in PhoenixSim using **PhotonicMessages**, which contains a pointer to a *PacketStat* object. This object tracks various physical-layer characteristics such as insertion loss and crosstalk.

Insertion Loss

Insertion loss is the power attenuation that an optical signal receives as it moves through a network. Loss can be attributed to multiple sources dealing with propagation or interactions with devices around the network. Losses are currently categorized into five areas: propagation loss, crossing loss, loss due to passing by a ring, loss due to passing into a ring, and bending loss. Crossing loss deals with the attenuation that is caused by waveguide crossings. Losses to passing by and passing into a ring deal with the attenuation of the signal that is caused by imperfect extinction of the signal. Within simulation, the total insertion loss contribution to a message is added to the *PacketStat* object each time the head of a message enters a *BasicElement* device.

Crosstalk

Crosstalk is any unintended noise power that is added to a message, which will result in a distorted signal being detected. This distortion can result in bit errors and are an inevitable part of the system. Due to the occurrence of such errors, a system must be in place to detect and fix these errors (either through error correction or retransmission).

The current crosstalk model accounts for two sources of noise power. The first source is called laser noise and arises due to inherent quantum and mechanical fluctuations in the optical signal of a laser. Another contributing factor to laser noise is the extinction ratio of the modulation technique, which determines how well a '1' can be distinguished from a '0'. We refer to the second source of noise and inter-message crosstalk, which arises imperfect coupling of optical signals in a photonic device. An example of this is a waveguide crossing, which is an inevitable consequence of the planar nature of on-chip topologies. Due to the physical intersection of two waveguides, situations can occur where two messages overlap at the crossing, directly leaking power onto each other. Similarly, imperfections in ring resonators can cause a non-ideal extinction ratio, allowing signals

to leak onto foreign messages.

A third form of crosstalk noise that we have planned to implement is intra-message crosstalk. This is a result of the WDM nature of many optical transmissions. When a WDM signal must be translated back into an electrical signal, each wavelength of the WDM message must be individually decoded. In order to extract each wavelength, a device must be present at the receiver that can 'select' the appropriate part of the message. This is usually implemented as a filter, however because filters do not necessarily completely block out other wavelengths, some noise power will inevitably get through. This can be particularly hazardous if there are many wavelength channels present in the transmission.

3.3 IO Plane

This section describes some of things that go on with regards to our current DRAM models: DRAMsim, and DRAM-LRL.

3.3.1 DRAMsim

DRAMsim is meant to be used for networks that support small memory transactions. In contemporary computing systems, a DRAM transaction is the size of a cacheline. Not very big. The typically DRAM subsystem is designed around this fact, namely in the memory controller and the shared bus-based interconnect between memory modules and memory controller.

Still, an application could request a large chunk of data from memory. The best way to show you what happens is to look at an example. Figure 3.4 shows the event diagram for a read transaction.

The transaction originates at a NIF somewhere in the network. A message containing a **ProcessorData** with type *cacheReadFromMemory* transmits across it as usual until it reaches the NIF attached to the *cDramModule*, which contains an instantiation of DRAMsim. The NIF goes through the usual protocol of interacting with a processor-like entity, which *cDramModule* also implements. In this example, the Transaction Queue in DRAMsim is full, and the *cDramModule* does not issue the send grant until it clears up. The **ProcessorData** eventually reaches the *cDramModule*. In this example, the read request was for twice the size of the cacheline size set in DRAMsim. This means that the original request must be broken into two DRAMsim transactions. This process enables application-level messages to and from DRAM of arbitrary size. After a DRAMsim transaction has been issued to DRAMsim, in this case of type *MemRead*, DRAMsim does its thing (which I won't get into here. You can ask DRAMsim's authors about that.). When the transaction is complete, DRAMsim calls the callback function *return_transaction(...)* in *cDramModule*. If the transaction was a read, as in this example, it then forwards it on to the NIF to be returned to the requester. A write is just ignored. And there you have it.

3.3.2 DRAM-LRL

For DRAM-LRL, we implemented our own memory controller and DRAM device models. DRAM-LRL is also a little more simple than DRAMsim, because it's made to function with a circuit-switched network, which vastly simplifies the controls. So we know and can describe all the nitty-gritties that go on. Again, let's go over an example. Figure 3.5 shows a write transaction.

A write begins with a NIF sending a *path_setup* to the destination MAP (memory access point). The NIF there receives it, and in this example, returns a *path_blocked* because the memory controller is currently servicing a read. This is necessary to avoid deadlock, or at least starvation,

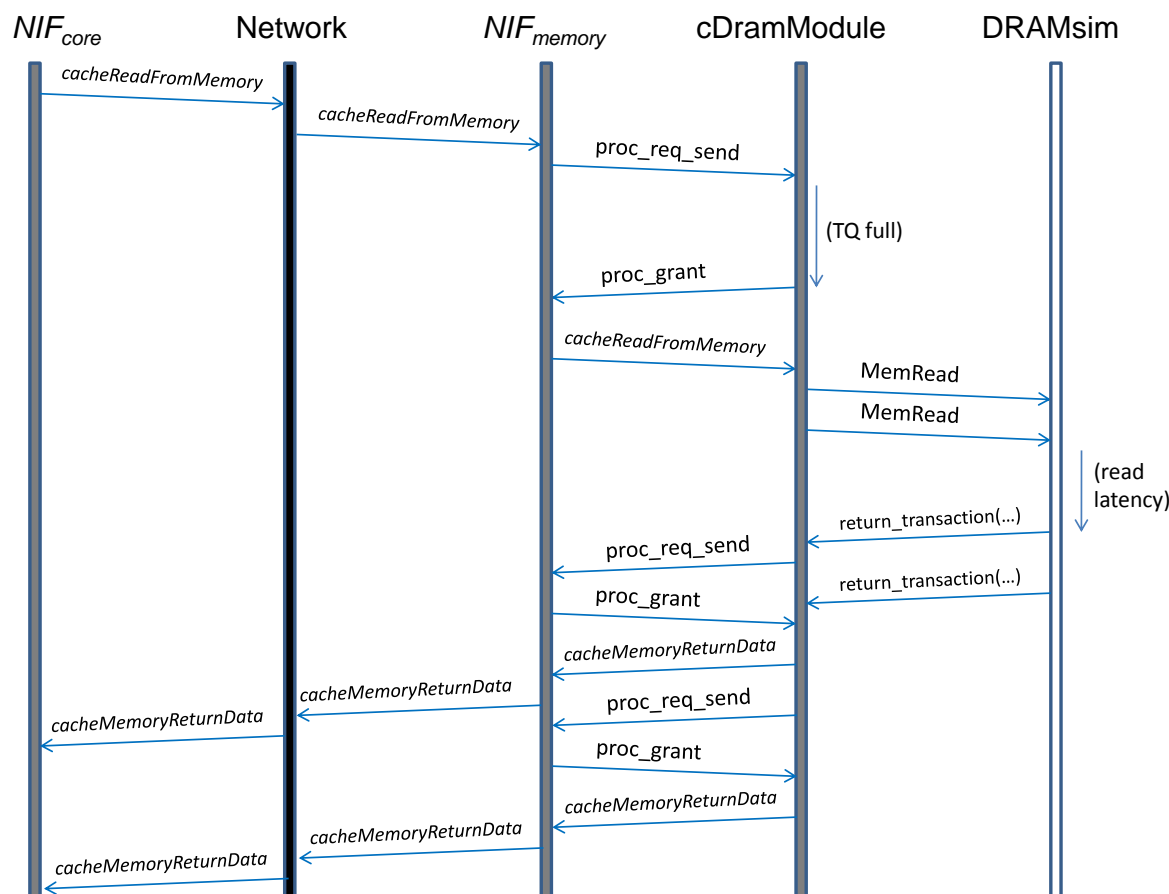


Figure 3.4: Read transaction example with DRAMsim.

because it releases the network resources that were reserved from the path setup getting there. After backoff, the original NIF retries the setup, and succeeds. The memory request is forwarded to the *MemoryControl* module. This guy issues the appropriate DRAM commands to the DRAM module (optically, setting the MAP switch first, which is not shown), sets the MAP switch for access to the network, and tells the NIF to send a *path_ACK* back to the NIF. This message also includes the *amount* of data that it is ok to send (at max, the size of the DRAM array's row). In this example, it takes two DRAM row transactions to write the entire message. This is accomplished by the NIF requesting that it send more, which it does by sending the memory controller a *requestDataTx* message. The memory controller issues more DRAM commands, and returns a *grantDataTx* message. The NIF then sends the rest of the data, and tears down the path. This protocol seems kind of unnecessary, but ensures that the memory controller is in charge of the DRAM timing, which is kind of critical.

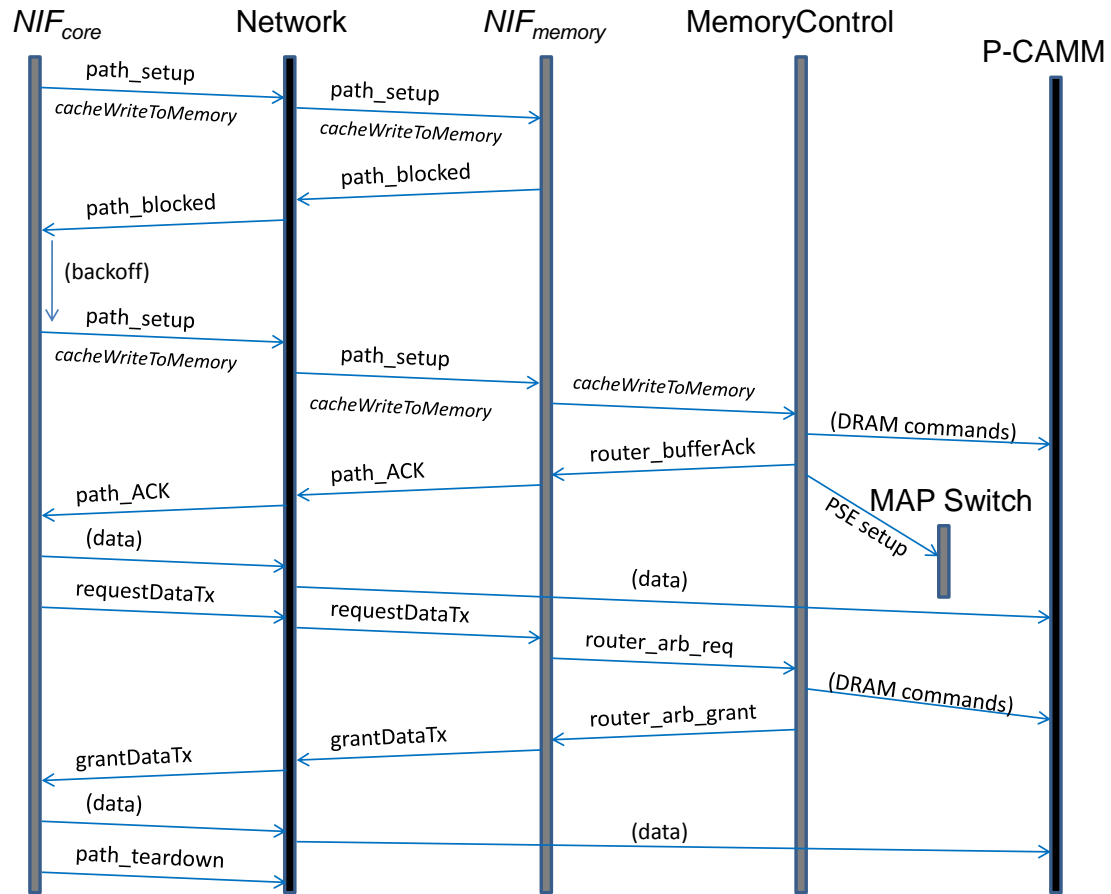


Figure 3.5: Read transaction example with DRAMsim.

Chapter 4

Network Design

This chapter describes some things to think about when designing the networks similar to the ones we have already modeled in PhoenixSim.

4.1 Electronic Networks

PhoenixSim was developed mainly to investigate photonic networks, as we've spent a lot of time developing an accurate photonic device library. However, it's often useful to be able to model electronic networks, either as a baseline for comparison, or as a part of a photonic network (such as a control plane). Here we briefly describe what you need to know about setting up your electronic network the way you intended.

4.1.1 Electronic Router

No doubt you will want to use the `ElectronicRouter` module we have provided with PhoenixSim to construct your electronic network (the `HybridRouter` module also uses this, for a control plane). There are a few important parameters to the `ElectronicRouter` module you should be aware of:

- `numPorts` - (int) - exactly what it sounds like. We assume number of inports == number of outputs.
- `numPSEports` - (int) - when an electronic router is connected to a photonic switch, it has to know how many rings it controls. We'll get this out of there in the future and have a better hierarchy of modules. Deal with it for now, just put 0 if there's no photonic switch.
- `type` - (int) - specifies the type of *Arbiter* to use. Types can be found in `electronicComponents/RouterArbiter.h`. See Section 6.3 for details on writing your own arbiter.
- `id` - (string) - specifies the network-level part of the router's address. See Section 2.6 for more info on addressing. If you're going with the standard "NET.MEM.PROC." addressing format, then this value will be something like

$$id = string(index) + "."; \tag{4.1}$$

where *index* is a NED keyword that refers to the index of the module when you are instantiating an array.

- level - (string) - again, dealing with addressing. This specifies which level in the address format this router belongs to. For the standard "NET.MEM.PROC." format, a router's level would be "NET";
- numX/numY - (int) - gives the router some info about how many nodes are in the network, for routing purposes.

4.1.2 Electronic Channels

Now that you have some routers, you'll need to hook them up. The *ElectronicChannel* module models many electronic wires in parallel, for one logical channel. These wires are optimally-repeated using ORION, based on their length. For now, we don't have a good way of specifying the exact layout of all wires in the network. So, the length of any wire can be specified with two parameters: *spaceLengths*, and *routerLengths*. Figure 4.1 illustrates these two parameters.

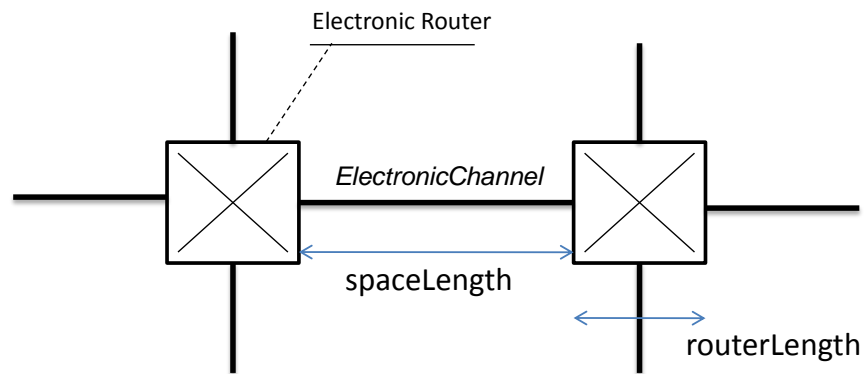


Figure 4.1: Two parameters for specifying the lengths of channels: *spaceLength*, *routerLength*.

Space lengths refers to the distance between two electronic routers, assuming that there is one router per core. This value is calculated from the *coreSizeX* parameter (see below), and a router's area (from ORION). Router length is the square root of a router's area (from ORION), assuming square routers. The default *ElectronicChannel* is *spaceLength* = 1, *routerLength* = 0. This allows us to easily use them in NED files when connecting routers together, such as

```
node[i].portIn[2] <-- ElectronicChannel <-- node[i+1].portOut[0];
node[i].portOut[2] --> ElectronicChannel --> node[i+1].portIn[0];
```

where putting "ElectronicChannel" between connecting nodes automatically instantiates an *ElectronicChannel* module with default values. Of course, to do more interesting things with wires which skip over one or more cores, we have to manually instantiate the *ElectronicChannels* and set their *spaceLength* and *routerLength* parameters accordingly.

4.1.3 Top-Level Parameters

There are some simulation parameters you should be aware of also, which you should make sure are set correctly in your .ini file.

- clockRate.cntrl - usually refers to the clock frequency of the wires.

- `electronicChannelWidth` - number of parallel wires in a logical electronic channel.
- `routerBufferSize` - size, in bits, of a single virtual channel of an inport.
- `routerVirtualChannels` - number of virtual channels. See Section 2.4.1 for details on how VC's work.
- `routerMaxGrants` - number of grants an arbiter can issue in one cycle. For electronic routers that are control routers for a circuit-switched network, make sure this is set to 1.

4.2 Circuit-switched Photonic Networks

LIGHTWAVE RESEARCH LABORATORY has been studying circuit-switched photonic networks since 2006 [2, 9, 16, 18–20]. This concept relies on a light-weight electronic control network to set up end-to-end optical paths between cores and memory.

Several limitations of optics prevent optical networks from being used in an identical manner to electronic networks. Traditional packet-switched electronic networks consist of many routers that each read in and route the data packets towards its intended destination. Optical buffering and in-flight all-optical processing are not seen as feasible technologies in the foreseeable future. In telecom- and metro-scale networks, the workaround is to do optical-electronic-optical (O-E-O) conversion at the expendable costs of small amounts of latency and increased power dissipation. At the chip-scale however, the latency penalty becomes relatively much more costly and the power dissipation penalty becomes ever so much more severe, therefore alternative approaches are necessary. To work around these limitations, alternative approaches are currently being investigated and predominately fall into two categories: wavelength-selective switching and spatial switching.

Wavelength-selective switching typically requires the predetermination of a path through a topology that completely specifies all routes through the network by the wavelength of the packet. Such topologies consist of carefully tuned filters that direct the optical message according to the wavelength of light that the message is encoded on. Ring filters can be designed to produce this exact functionality. Figure 4.2a shows the required alignment for a single wavelength to be filtered out. Figure 4.2c shows how the optical path is altered according to whether the wavelength of light is on or off resonance with the filter. An optical packet can be delivered all the way to its destination using this technique without requiring any O-E-O conversion, simply by determining the entire network route at the transmitting node. This is advantageous to other photonic on-chip techniques since it results in superior latency performance, but can be disadvantageous in terms of spectral efficiency for transmission throughput.

Spatially switched networks use broadband WDM signaling to produce extremely high-bandwidth connections between communicating nodes by multiplexing data onto many parallel wavelengths. Switching is enabled through the use of broadband switches that can actively be controlled to pass or switch all the wavelengths concurrently. Ring resonators, given their versatile nature, can also be designed in this manner. Figure 4.2b shows how a ring resonator's resonant profile can be designed to align with many optical wavelengths at once. If we wanted to alter the state of the ring to ignore each wavelength, we would just need to shift the profile to the left or right which effectively detunes the ring. This can be done in rings through either thermal or electro-optic means. Similarly to the wavelength selective case, the optical message can no longer be controlled like the examples shown in Figure 4.2c. However instead of the wavelength dictating the propagation behavior, it is

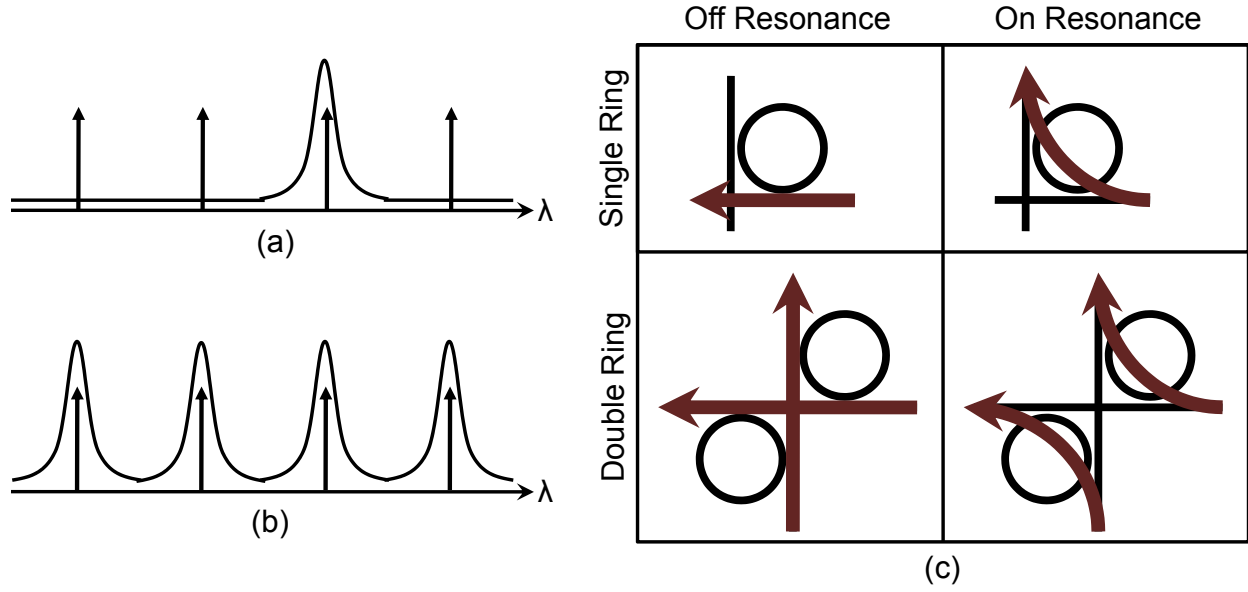


Figure 4.2: Optical ring resonators can be used as switching devices within an on-chip photonic network. (a) Ring resonators can be designed to behave like filters (shown as a spectral profile), picking out a pre-determined wavelength (drawn as an arrow) in a multi-wavelength communication system. (b) Rings can also be designed to interact with many wavelength concurrently, creating a broadband like device. (c) Two examples of switching elements designed with ring resonators is illustrated. When a wavelength is not in-tune with the passband of the rings (off resonance), an optical signal will pass by the ring uninterrupted. Wavelengths of light that are aligned the the rings passbands (on resonance), will couple into the ring and later the path of the message.

now determined by the switch itself. Since this method requires some sort of logical control to activate or de-activate the ring, separate electronic control circuitry is required. Since repeated O-E-O conversion at every ring that needs switching is detrimental to the cost of this sort of system, we instead design another dedicated low-bandwidth electronic control plane to perform network arbitration.

4.2.1 Insertion Loss - Bandwidth tradeoff

A key cost-benefit relationship that arises with photonic networks that does not exist in electronic networks is the notion of insertion loss vs. performance. Figure 4.3 summarizes the relationship between the insertion loss performance of a network and the extent that WDM can be used as a result of this limitation.

A key factor in this relationship is due to the optical power budget, which dictates the range of powers that can be sustained within the photonic devices due to fundamental limitations of the technology. The upper limit in power is due to nonlinear effects that can arise in the photonic devices. At great enough injected powers, the nonlinearities can cause higher losses and self modulation at ring resonator locations. To avoid this issue, we can place a limit in the injected power we allow into the system. At the lower end of the power range is the detector sensitivity. For the detectors to be able to reliably receive the signal, a certain threshold of photons needs to reach the detector. This can be achieved by dictating a lower limit in optical power that can be used in the network. These two limits, the nonlinear threshold and the detector sensitivity, form

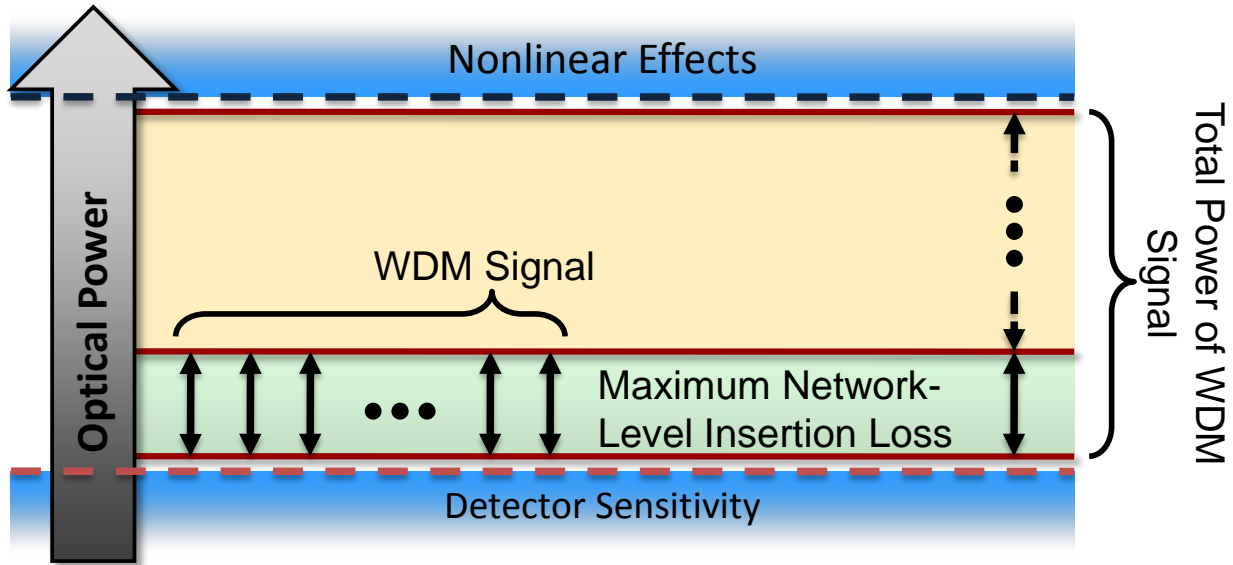


Figure 4.3: Insertion loss vs. bandwidth tradeoff. The network-level insertion loss and the optical power budget are intertwined in influencing the scalability and performance of the network. They dictate how large and complex a network can become and also determine the number of wavelengths that can be used by the network.

the optical power budget which will mandate how many wavelengths can be used and how far each optical signal can travel.

When considering the nonlinear threshold, a designer should note that the limit is placed on the total power in the waveguide and not on individual wavelengths. Therefore in the case of a WDM signal, the total power would be based on the aggregate power of all the individual wavelengths. This will further lower the injected power that each individual wavelength can be used at. Lastly, once we have determined the maximum power each wavelength can be injected at, we must account for the losses in the network and ensure that the optical signal that reaches the detector is at a sufficient power level.

By considering all these relationships, we see a tradeoff that exists between two system-level traits. If we allow a higher worst-case insertion loss to exist in our network design, we will lower the limit in the number of WDM channels that can exist. At the same time, if we were to produce networks with lower (better) insertion loss characteristics, we can increase the number of WDM channels used in the network. From a system standpoint, the number of WDM channels used in the network linearly relates to the bandwidth that a communication channel can achieve, since we can assume the multiplexing of data. At the same time, the insertion loss characteristics of a network largely relate to the complexity and size of the network. A network with a large number of nodes will likely result in much higher insertion losses than a network with a small number of nodes. This tradeoff becomes a key consideration that must account for the requirements and specifications of the system being designed for, and can also be used as arguments for driving certain device development in order to obtain better performance.

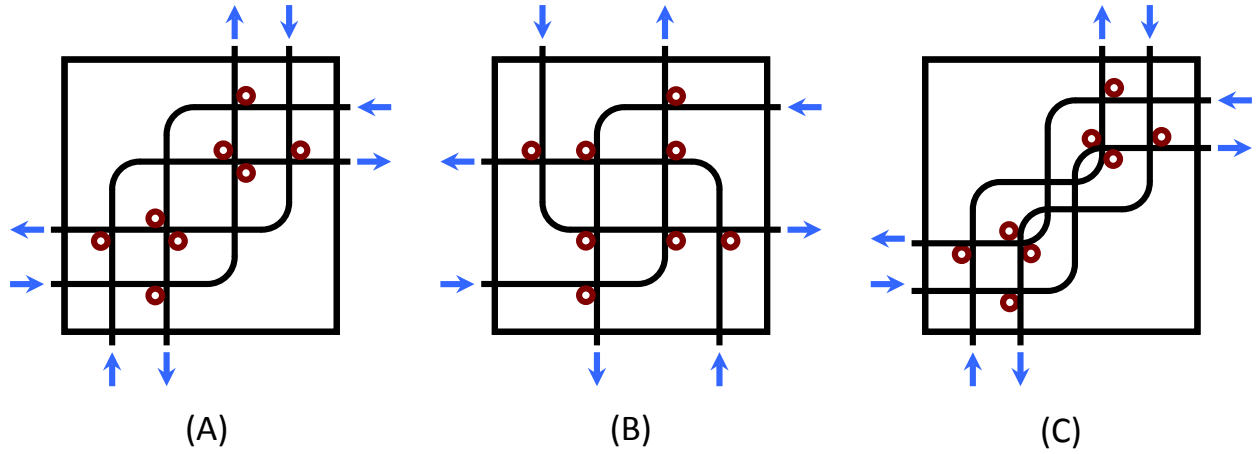


Figure 4.4: Various 4×4 non-blocking switch designs. (A) the original 4×4 non-blocking switch, (B) a 4×4 non-blocking switch that reduces the number of crossings, and (C) a 4×4 non-blocking switch that minimizes insertion loss for the straight path cases (North \rightarrow South, and West \rightarrow East).

4.2.2 4×4 Switch Designs

The 4×4 switch is a fundamental component in a variety of photonic topologies. There are several considerations that need to be made in a well designed 4×4 switch. First is to minimize the number of rings required which will reduce the amount of power required to control and thermally tune the devices. Second, is to minimize the insertion loss requirements of the design. Initial work concluded with the design shown in Figure 4.4A. Later, new 4×4 non-blocking switch designs were developed that optimized for various network cases (Figure 4.4B and C).

Chapter 5

Current Networks

This chapter describes the current network topologies that are included in the PhoenixSim distribution. They can be used immediately as sample networks, or can be used as templates for creating your own networks. The included networks are roughly categorized into three types: 1) electronic networks which networks implemented using standard router pipeline modeling, 2) a class of photonic networks that have been extensively studied in the LIGHTWAVE RESEARCH LABORATORY that uses circuit-switching for arbitration, 3) and miscellaneous photonic networks that have been proposed by groups outside of the LIGHTWAVE RESEARCH LABORATORY.

5.1 Electronic Networks

Electronic NoCs are the near-term solution to interconnecting many cores in a CMP because they are already established in the CMOS production line. Aside from their simplicity, many of these electronic topologies are used for applications that have traffic patterns that can be well mapped, allowing better utilization of network resources. All topologies assume a regularly spaced grid configuration. For the electronic networks illustrated below, red circles represent the communicating nodes, while the blue squares represent network routers.

5.1.1 Electronic Mesh

The electronic mesh topology connects the nearest neighbor nodes along each of the four cardinal directions (N, S, E, and W). This formation is well suited for a 2-D topology since it contains now wire crossings, thereby reducing the complexity of the chip.

5.1.2 Electronic Mesh - Circuit Switched

Conventionally, meshes are used as packet-switched networks, however the design can be adapted to use a circuit-switching protocol. A circuit-switched electronic mesh effectively requires two separate mesh networks, one to transmit control and arbitration packets in a packet switched manner. This packet-switched plane is used to configure a secondary data plane that is configured to be purely circuit switched for transmitting large data payloads across the network without buffering. This can produce a significant reduction in power dissipation since only the control message need to be buffered, while the payload message can propagate through the network without any buffering. This can also possibly reduce latency since once a path is setup, no clock cycles are expended for data to be buffered along the circuit path.

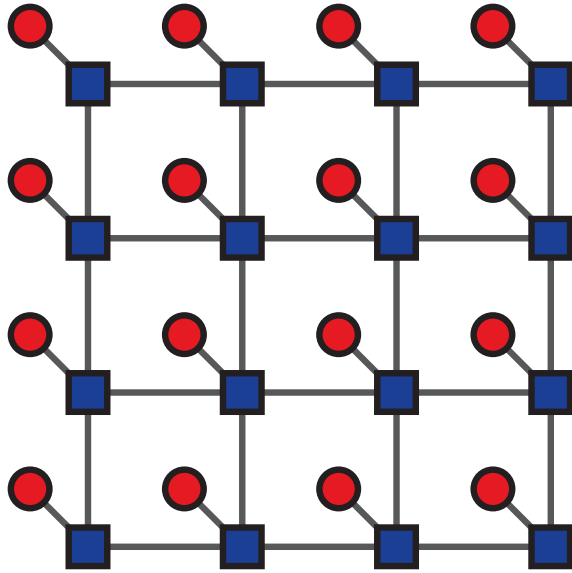


Figure 5.1: Electronic mesh topology.

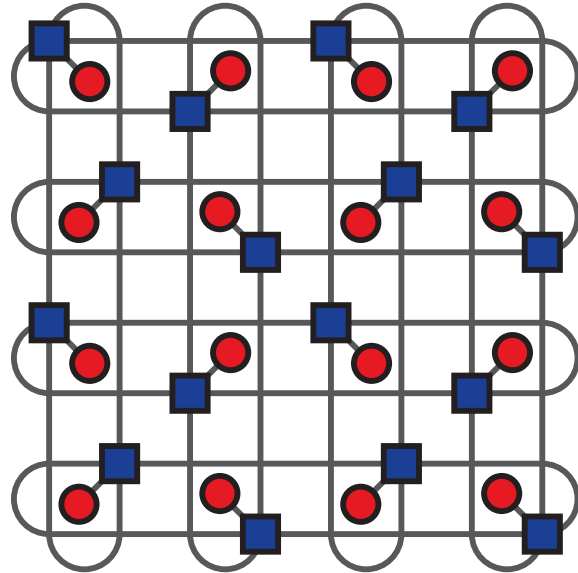


Figure 5.2: Electronic torus topology.

5.1.3 Electronic Torus

In a mesh topology, the nodes at the edges of the network will have one or two fewer connections than a node in the middle. In many circumstances, each 'tile' of the network is designed to be exactly the same (e.g. a five-port router that connects N, S, E, W, and the node). For the nodes at the end this can leave one or more dangling connections that are unused and wasted. The torus topology can take advantage of these by taking every pair of edge nodes along a single row or column and connecting their un-connected ports. This produces a ring along each row and column, and increasing the diversity of the network. The one disadvantage to this approach is that the connections that connect the edges together can be very long.

Figure 5.2 shows a further optimization that can be implemented by altering the positions of the routers, referred to as a *folded torus*. This eliminates the long wires by connecting the second-nearest neighbors and enables a network that can be traversed in fewer hops as well.

5.1.4 CMesh

Similar to the torus topology, CMesh takes advantage of unconnected ports by connecting them to other routers along the same edge [1]. In contrast with the torus, no additional wire crossings are introduced since the added connections remain around the edge of the chip. A version of a 4×4 CMesh is illustrated in Figure 5.3.

5.1.5 Flattened Butterfly

The Flattened Butterfly attempts to decrease transmission latency in the network by reducing the number of hops required to travel from any source-destination pair. This form of network takes advantage of high-radix routers (routers with many ports) so that a packet can travel anywhere along the length of a row or column in a single hop [11]. In effect, this allows any packet to move from source to destination in two hops (assuming dimension ordered routing).

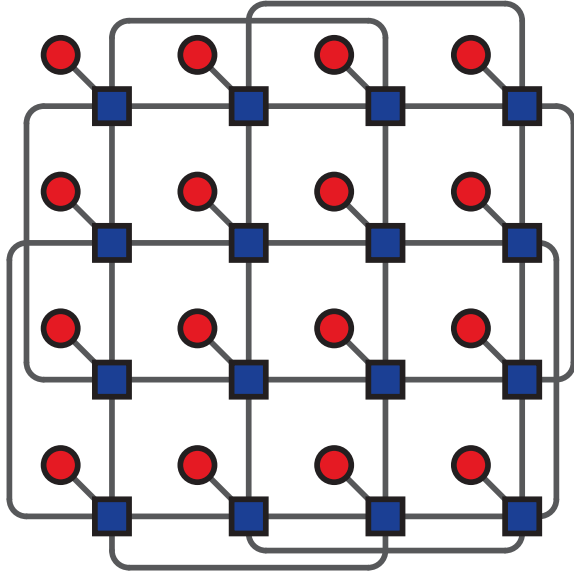


Figure 5.3: Electronic CMesh topology.

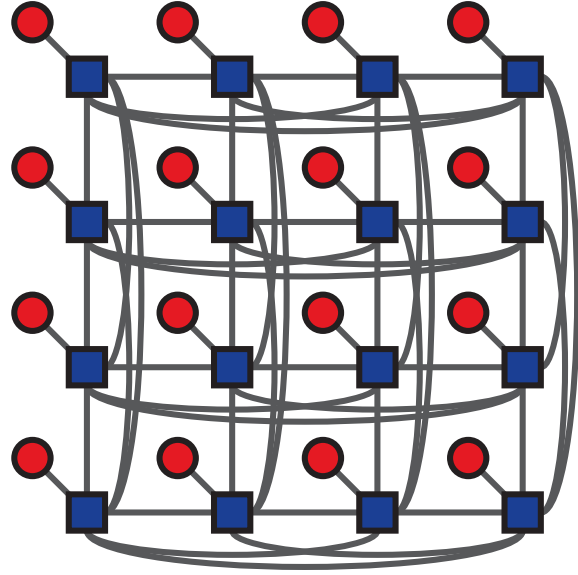


Figure 5.4: Electronic Flattened-Butterfly topology.

5.2 Hybrid Circuit-Switched Photonic Networks

The remainder of this section is left to describing several hybrid photonic networks that have been devised and published by the LIGHTWAVE RESEARCH LABORATORY group.

5.2.1 Photonic Torus

The photonic torus was the first proposed optical on-chip network to use the concept of spatial switching [18]. This topology, shown in Figure 5.5 is meant to mimic the layout of a folded-torus network. The illustration shows the torus as the bold black lines, with the switching nodes indicated by blocks marked with an 'X'. The switching nodes were only designed as 4×4 photonic switches, therefore cannot be easily modified to have a fifth connection to a communication end-point. Instead, this design uses an complimentary access network (shown as red lines and red blocks) to allow messages to enter and exit the torus. 'G' marks the end-point where messages enter and exit the topology, while 'I' and 'E' mark the optical switches used to inject and eject from the torus, respectively.

5.2.2 Nonblocking Torus

The nonblocking version of the photonic torus is illustrated in Figure 5.6, first designed in [16]. This version of the network attempts to increase performance by creating a strictly non-blocking network, which allows a source to be able to allocate a path on the network to any destination as long as the destination is not busy with another transmission. This is not true of the original photonic torus since paths in the network can be allocated in a way that will block other transmissions from occurring.

5.2.3 Photonic Mesh

The photonic mesh is characteristically similar to an electronic mesh, which has been designed to use 5×5 photonic switches. See [8] for more details. While this design might decrease the path

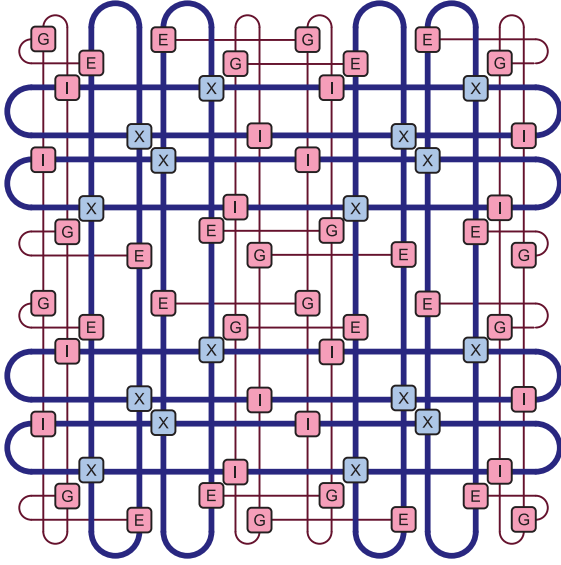


Figure 5.5: Photonic Torus topology.

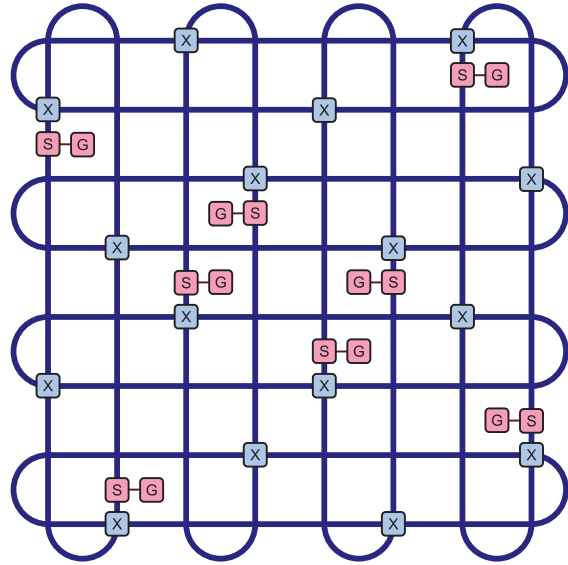


Figure 5.6: Photonic Non-blocking Torus topology.

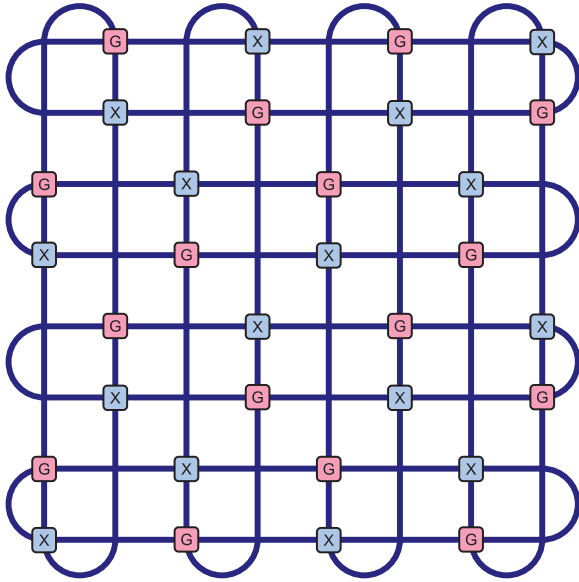


Figure 5.7: Photonic TorusNX topology.

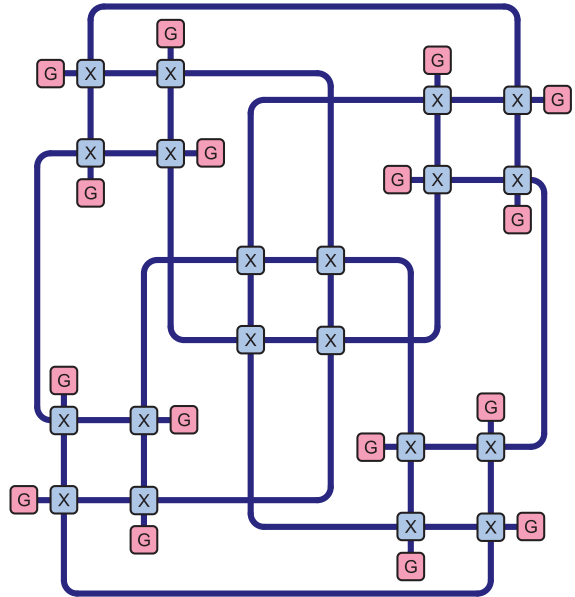


Figure 5.8: Photonic Square Root topology.

diversity that can be offered to a packet, it makes up for it by lowering the insertion loss costs considerably.

5.2.4 TorusNX

TorusNX is a newly developed network that simplifies the photonic torus which results in more efficient layout, better physical-layer performance, and better system-level performance.

5.2.5 Square Root

Square Root takes a non-traditional approach to the photonic topology by considering a hierarchical layout (shown in Figure 5.8).

Chapter 6

How To ...

This section describes some of things that we typically do, and we assume that you might too. Actually, this section is actually more a quick resource guide. There are plenty of examples in the code to guide you, so you shouldn't have much trouble.

Since you have the code, you can do whatever you want with it. However, we suggest you read these sections and follow them carefully to use the simulator as we intended it to be used (*i.e.* correctly). If you are planning on branching out and doing things your own way, go right ahead, but make sure you have a good grasp of the whole simulator.

6.1 Create my own photonic device

Section 2.4.2 covered the current types of devices which can be modeled in PhoenixSim currently. Creating a new abstract class is beyond the scope of this particular How To. To create a new device, we typically treat it as a black box device, we do not concern ourselves with the actual layout or geometry of the waveguides and doping regions, but more concerned about what an optical signal will see as it enters or exits the device from its various ports.

I will use the `pse1x2` device as an example. First look at the NED file. We know this is a ring resonator structure and because it exhibits multiple states due to electro-optics (therefore we use the `ActiveRingElement` base class) we know this will be a two state device (`int numOfStates = 2;`). Next we designed this particular switching to include two orthogonal waveguides, with a ring resonator at the intersection, therefore there will be four ports (`int numOfPorts = 4;`). This particular design also only uses a single ring (`int numOfRings = default(1);`). The remaining parameters simply serve to enable parameters to be passed into the C++ code. Also, we must declare the names of the four ports (`photonicHorizIn`, `photonicHorizOut`, `photonicVertIn`, `photonicVertOut`), and the control port (`fromRouter`) which will control what state the device is in.

Next we go into an overview of how the header and source file are put together. Because this class inherits from the `ActiveRingElement` class, which itself inherits from the `BasicElement` class, a lot of the underlying algorithms are already taken care of, and only a few specific things need to be written to describe the actual device. First is the `Setup()` function, which must be implemented to organize the gates. Essentially `gateIdIn` and `gateIdOut` should be assigned to the input and output versions of the gates that were defined in the NED file. Matching indices of the two arrays should point to the same gate, only differing whether it is referring to the input or output. Also `gateIdFromRouter` should point to the `gateId` of the input gate that will be used for control. The remaining code you see in the `pse1x2` example are not necessary but are useful when we need to pull in the variables that were mentioned from the NED file.

In `HandleControlMessage()`, we see that this is where control messages from `gateIdFromRouter` will be handled. Essentially, the messages that come in will store a number corresponding to the state that the device should transition to. In `GetLatency()`, you are passed as arguments the index of the input and output gate and the function should return the latency of that input/output combination. `GetPropagationLoss()`, `GetCrossingLoss()`, `GetDropIntoRingLoss()`, `GetPassByRingLoss()` each are declared to enable losses in those categories. If any were not declared (in this case `GetBendingLoss()` is not declared) then the base class's implementation returns a default 0 dB. `GetPowerLevel()` should return the static power dissipation in a particular state of the device. `GetEnergyDissipation()` specifies the dynamic energy dissipation for a state transition. `SetRoutingTable()` specifies the output gate for which an input gate should propagate to. This should be specified for each `currState` if they are different. In the case of ring resonator devices, wavelengths can also be off-resonance, therefore `SetOffResonanceRoutingTable()` also needs to be set for optical signals that are not on resonance.

6.2 Create my own Network Interface (NIF)

A new *NIF* is implemented by inheriting from the abstract *NIF* class, in `processingPlane/interfaces/NIF.h`, or one of its subclasses, such as *NIF_Packet_Credit* or *NIF_Circuit*. The *NIF* class is only aware of how to interact with the processor-side, as explained in Section 3.1.2.

Your new *NIF* will inherit from one of the abstract classes mentioned, and implement the pure virtual functions necessary which define how to communicate with the network attached to it. Also, you'll need to create the appropriate .NED file, which will likely be exactly like the other *NIFs*. But that's the way OMNeT works.

See Section 2.3.1 for details on how the *NIF* generally works.

6.3 Create my own electronic router

You can use our electronic router model for the usual 3-stage pipelined router: input, arbitration, crossbar traversal. To do this, you basically only need to write your own *Arbiter* class, which mainly just implements the `route(...)` function. See `electronicComponents/arbiters/Arbiter_EM.cc` for an example. If you want to do more tricky things with hierarchical routing, you'll need to mess with the address format, which may require implementing the `getUpPort(...)` or `getDownPort(...)` functions (see Section 2.6). Once you have your new arbiter, just set the *type* parameter in `ElectronicRouter` to the id you assigned to your new router, and that's it.

6.4 Create my own hybrid router

As you recall from Section 2.4.3, the `HybridRouter` module is used in circuit-switched networks, which consists of an electronic router and a photonic switch. Read Section 6.3 right above for information on the electronic router. Section 4.2 might be useful for photonic switch design, which must implement the `PhotonicSwitch` module interface. Then just set the `optSwitch` parameter of `HybridRouter` to the name of your switch, and you're good to go.

6.5 Create my own network/topology

Ah, here it is. This is why you're reading this whole darn document. You want to create a network. Honestly, the easiest way to go about it is copy an existing one. Try `topologies/electronicMesh/electronicmeshnetwork.ned` for a simple one. The modules you need to instantiate:

- Statistics module - so that you can measure things

- Processing plane - there are two versions, one that uses a processor router, and one which relies on network-side concentration. See Section 2.3 for details.

The rest is up to you. We usually make one or more logical planes for the network components, and hook them up in the top level network .ned file. You'll also want to instantiate an instance of IOPlane if you want to use the DRAM models.

6.6 Run my own application/application model

You'll be making a subclass of *Application*, so first read Sections 2.3.2 and 3.1.1 to get an idea of how the *Application* class interacts with the *Processor*. You basically have the option of implementing 4 functions. How you do this depends on the application, just make sure you understand under what circumstances those functions are called by *Processor*. Also, you can use the 5 general parameters at your disposal (param1, param2, param3, param4, param5) which are a part of the *Application* class. Once you have your subclass, you just have to add it to *Processor*'s instantiate(...) method with the other applications so it can get instantiated. You'll have to make up a name for it that you can pass through the application parameter.

6.7 Measure something with custom statistics

So you want to measure something about your network? First, read Section 2.7 to get a feel for the classes we've provided. Ok, now that you're an expert on the subject, all you have to do is call Statistics::registerStat(...) in the initialize(..) function of your module, save the StatObject pointer that it returns, and call statObj→track(...) when you want to record something. And voila, at the end of the simulation it will print it to the file just like you expected.

If for some reason we don't have a mechanism that measures something you want, you can just create new StatObject, StatGroup, or LogFile subclasses to do exactly what you want. Remember that a StatObject defines how something is measured, a StatGroup defines how StatObjects are collected together and reported, and a LogFile is just a collection of StatGroups.

Chapter 7

Appendix

This chapter lists some things that might be useful.

7.1 List of all OMNeT Parameters

This section lists all of the parameters passed through OMNeT to the simulator. Changing these don't require recompiling the simulator, and all the OMNeT parameter mechanisms apply.

Table 7.1: All OMNeT PhoenixSim Parameters

Parameter	Type	Description
<i>Electronic Parameters</i>		
clockRate_cntrl	int	Frequency of the electronic plane, in Hz.
electronicChannelWidth	int	Number of bits (wires) of a single electronic channel.
maxPacketSize	int	Max size of a packet in electronic network, in bits.
routerBufferSize	int	Size of a single virtual channel's buffer in one electronic router port, in bits.
routerMaxGrants	int	Number of grants an electronic arbiter can give in one cycle.
routerVirtualChannels	int	Number of virtual channels in electronic router.
virtualChannelControl	int	Selects the policy for virtual channel assignment. Use 0 for now.
wireDoublePumping	bool	True if inter-router electronic wires are double-pumped.
<i>Common Optical Parameters</i>		
clockRate_data	int	?
firstWavelength	double	when useWDM is true, represents the first wavelength in a series of wavelengths that forms an entire WDM signal.
groupLabel	string	Used for statistics gathering, the devices with common names will have data reported as an aggregate value (e.g. sum of all energies dissipated by a modulator across an chip).
Switch_FreeSpectralRange	double	Free spectral range of switches.
Continued on next page		

Table 7.1 – continued from previous page

Parameter	Type	Description
useWDM	bool	Resonant wavelengths represented as an array of values when enabled.
wavelengthSpacing	double	Represents the spacing between wavelengths when useWDM is enabled.
<i>Specific Optical Parameters</i>		
BarDelay_2x2	double	Delay for an optical signal when the 2x2 switch is in a bar state.
BendingLoss	double	Insertion loss for bends (excluding propagation).
CrossDelay_2x2	double	Delay for an optical signal when the 2x2 switch is in a cross state.
CrossingLoss	double	Insertion loss due to waveguide crossings (excluding propagation).
Crosstalk_Cross	double	Crosstalk due to waveguide crossings.
Detector_EperBit	double	Energy per bit to detect a single bit.
DetectorSensitivity	double	Sensitivity of the detector.
DropDelay_1x2	double	Delay for dropping into ring, in a 1x2.
DropDelay_1x2NX	double	Delay for dropping into a ring, in a 1x2NX.
DropDelay_Filter1x2	double	Delay for dropping into a ring filter.
IL_Facet_Lateral	double	Loss due to a facet using lateral coupling (excluding propagation).
IL_Facet_Vertical	double	Loss due to a facet using vertical coupling (excluding propagation).
IntraPacketTime	double	??? might not exist anymore ???
LaserEfficiency	???	???
LaserPower	double	Power level of laser output.
LaserRelativeIntensityNoise	double	Relative intensity noise level of laser.
Latency_Bend	double	Latency through a 90 degree bend.
Latency_Cross	double	Latency through a 50 um crossing, with double-etched region.
Latency_Detector	double	Latency through a detector.
Latency_Modulator	double	Latency through a modulator.
LatencyRate_Line	double	Latency rate of silicon waveguides.
ModulatorExtinctionRatio	double	Extinction ratio of modulators.
Modulator_EperBit	double	Dynamic energy per bit for modulators.
Modulator_Static	double	Static power dissipation for modulators.
PassByRingLoss	double	Loss for passing by a ring off resonance.
PassThroughRingLoss	double	Loss for dropping into a ring on resonance.
PathSeparation	double	In Torus and NB Torus, the separation between the bidirectional waveguides on the main loops of the tori.
PropagationLoss	double	Insertion loss rates of silicon waveguides.
RingDrop_ER_Filter1x2	double	The extinction ratio of the drop path
Continued on next page		

Table 7.1 – continued from previous page

Parameter	Type	Description
		through a 1x2 ring filter device.
RingDynamicOffOn	double	Dynamic energy dissipated when turning a broadband ring switch on.
RingDynamicOnOff	double	Dynamic energy dissipated when turning a broadband ring switch off.
RingOff_ER_1x2	double	The extinction ratio of the through path through a 1x2 ring switch device.
RingOn_ER_1x2	double	The extinction ratio of the drop path through a 1x2 ring switch device.
RingOff_ER_1x2NX	double	The extinction ratio of the through path through a 1x2 NX ring switch device.
RingOn_ER_1x2NX	double	The extinction ratio of the drop path through a 1x2 NX ring switch device.
RingOff_ER_2x2	double	The extinction ratio of the through path through a 2x2 ring switch device.
RingOn_ER_2x2	double	The extinction ratio of the drop path through a 2x2 ring switch device.
RingStaticDissipation	double	Static power dissipation for each ring in a broadband switch configuration.
RingThrough_ER_Filter1x2	double	The extinction ratio of the through path through a 1x2 ring filter device.
ShiftToRecenter	XXXX	??? Not a generic parameter. ???
thermalRingTuningPower	double	Power dissipation per degree per ring to compensate for thermal drift during operation of ring resonator devices.
thermalTemperatureDeviation	double	Average temperature deviation per ring.
ThroughDelay_1x2	double	Delay for through port on a 1x2 ring switch.
ThroughDelay_1x2NX	double	Delay for through port on a 1x2 NX ring switch.
ThroughDelay_Filter1x2	double	Delay for through port on a 1x2 ring filter.
TurnDistance	???	special value for Torus and NB Torus.
TurnDistanceEjection	???	special value for Torus and NB Torus.
TurnDistanceInjection	???	special value for Torus and NB Torus.
<i>DRAM Parameters</i>		
DRAM_config_file	string	Location of the config file used in DRAMsim.
DRAM_arrays	int	In DRAM-LRL, number of arrays per bank.
DRAM_chipsPerDIMM	int	In DRAM-LRL, number of DRAM chips per memory module.
DRAM_cols	int	In DRAM-LRL, number of columns in an array.
DRAM_freq	int	In DRAM-LRL, frequency of DRAM devices.
DRAM_rows	int	In DRAM-LRL, number of rows in an array.

Continued on next page

Table 7.1 – continued from previous page

Parameter	Type	Description
dramFrequency	int	DRAMsim frequency.
ioChannelWidth	int	Width of off-chip electronic channels, in number of wires.
offChipClockRate	int	Frequency of off-chip channels.
<i>Application Parameters</i>		
application	string	Name of the application to run. See processingPlane/Processor.cc.
appParam1	double	Application-specific.
appParam2	double	Application-specific.
appParam3	double	Application-specific.
appParam4	double	Application-specific.
appParam5	string	Application-specific.
procMsgOverhead	int	Overhead added to outgoing messages from <i>Processor</i> , in clock cycles.
<i>General Parameters</i>		
addressTranslator	string	Specifies how to translate core IDs to addresses. See Section 2.6 for more info.
coreSizeX	int	Width of one core, in μm .
coreSizeY	int	Height of one core, in μm .
customInfo	string	Info to append on to the results file name.
logDirectory	string	Directory to put the simulation results.
ioPlaneConfig	string	Specifies how cores are mapped to memory modules, and where memory modules are located in the network. See Section 2.5.3 for more details.
networkName	string	Name of the network, used in results file names.
networkProfile	string	Specifies the structure of addresses, see Section 2.6 for more info.
NIF_type	string	Name of the NIF to use.
numOfNodesX	int	Number of x-axis network nodes.
numOfNodesY	int	Number of y-axis network nodes.
processorConcentration	int	Number of cores that share a network gateway.
theSwitchVariant	int	Switches between different layouts of the same photonic switch.
useIOplane	bool	Indicates whether or not the simulation should include the IO plane.

7.2 ORION Parameters

This section lists some parameters in ORION, found in `electronicComponents/orion/ORION_2_Params/ORION_port.h`. Note that changing these does require recompiling the simulator. We only

list the most important ones, the rest in ORION_port.h you can refer to the ORION documentation [25].

Table 7.2: ORION Parameters

Parameter	Description
PARM_Tech_POINT	Technology point. For now, 32 is the lowest it goes.
PARM_TRANSISTOR_TYPE	Normal (NVT), High (HVT), or Low (LVT) threshold voltage.
PARM_Vdd	Chip voltage.
PARM_Freq	Frequency that the electronics run at.

Bibliography

- [1] J. Balfour and W. J. Dally. Design tradeoffs for tiled cmp on-chip networks. In ICS '06: Proceedings of the 20th annual international conference on Supercomputing, pages 187–198, New York, NY, USA, 2006. ACM.
- [2] J. Chan, A. Biberman, B. G. Lee, and K. Bergman. Insertion loss analysis in a photonic interconnection network for on-chip and off-chip communications. In IEEE Lasers and Electro-Optics Society (LEOS), Nov. 2008.
- [3] J. Chan, G. Hendry, A. Biberman, K. Bergman, and L. P. Carloni. Phoenixsim: A simulator for physical-layer analysis of chip-scale photonic interconnection networks. In DATE: Design, Automation, and Test in Europe., Mar. 2010.
- [4] L. Chen, K. Preston, S. Manipatruni, and M. Lipson. Integrated GHz silicon photonic interconnect with micrometer-scale modulators and detectors. Optics Express, 17(17), August 2009.
- [5] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. Mathematics of Computation, 19:297–301, 1965.
- [6] Corning Inc. Datasheet: Corning SMF-28e optical fiber product information. Online at <http://www.princetel.com/datasheets/SMF28e.pdf>.
- [7] M. Frigo and S. G. Johnson. benchFFT. Available online at <http://www.fftw.org/benchfft/>.
- [8] G. Hendry, J. Chan, D. Brunina, L. P. Carloni, and K. Bergman. Circuit-switched silicon nanophotonic networks-on-chip for ultra power-efficient off-chip memory access. In ACM/IEEE International Symposium Computer Architecture, 2010. in review.
- [9] G. Hendry et al. Analysis of photonic networks for a chip-multiprocessor using scientific applications. In The 3rd International Symposium on Networks-on-Chip, May 2009.
- [10] IBM Corporation. The cell project. Online at <http://www.research.ibm.com/cell/>.
- [11] J. Kim, J. Balfour, and W. Dally. Flattened butterfly topology for on-chip networks. Microarchitecture, IEEE/ACM International Symposium on, 0:172–182, 2007.
- [12] B. G. Lee, X. Chen, A. Biberman, X. Liu, I.-W. Hsieh, C.-Y. Chou, J. Dadap, R. M. Osgood, and K. Bergman. Ultra-high-bandwidth WDM signal integrity in silicon-on-insulator nanowire waveguides. IEEE Photonics Technology Letters, 20(6):398–400, May 2007.
- [13] B. G. Lee et al. High-speed 2×2 switch for multi-wavelength message routing in on-chip silicon photonic networks. In Eur. Conf. on Optical Commun., volume 2, Sept. 2008.
- [14] B. G. Lee et al. High-speed 2×2 switch for multiwavelength silicon-photonic networks-on-chip. Journal of Lightwave Technology, 27(14):2900–2907, July 2009.

- [15] B. E. Little et al. Ultra-compact Si-SiO₂ microring resonator optical channel dropping filters. IEEE Photonics Technology Lett., 10(4):549–551, Apr. 1998.
- [16] M. Petracca, B. G. Lee, K. Bergman, and L. Carloni. Design exploration of optical interconnection networks for chip multiprocessors. In 16th IEEE Symposium on High Performance Interconnects, Aug 2008.
- [17] V. Puente, R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, and C. Izu. Adaptive bubble router: a design to improve performance in torus networks. In Proc. Of International Conf. On Parallel Processing, pages 58–67, 1999.
- [18] A. Shacham, K. Bergman, and L. Carloni. On the design of a photonic network-on-chip. In First International Symposium on Networks-on-Chip, 2007.
- [19] A. Shacham, K. Bergman, and L. P. Carloni. Photonic networks-on-chip for future generations of chip multiprocessors. IEEE Transactions on Computers, 57(9):1246–1260, 2008.
- [20] A. Shacham et al. Photonic noc for dma communications in chip multiprocessors. In 15th Annual IEEE Symposium on High Performance Interconnects, Aug 2007.
- [21] Tiler Corporation. TILE-Gx processor family. Online at <http://www.tilera.com/products/TILE-Gx.php>.
- [22] A. Varga. OMNeT++ discrete event simulation system. <http://www.omnetpp.org>.
- [23] Y. Vlasov, W. M. J. Green, and F. Xia. High-throughput silicon nanophotonic wavelength-insensitive switch for on-chip optical networks. Nature Photonics, 2:242–246, Apr. 2008.
- [24] D. Wang et al. DRAMsim: A memory-system simulator. SIGARCH Computer Architecture News, 33(4):100–107, Sept. 2005.
- [25] H. Wang et al. ORION: A power-performance simulator for interconnection networks. In 35th International Symposium on Microarchitecture, 2002.
- [26] Q. Xu et al. 12.5 Gbit/s carrier-injection-based silicon micro-ring silicon modulators. Opt. Exp., 15(2):430–436, 2007.